From Mac to Windows

An Introduction to Microsoft Windows for Macintosh Programmers

Stephen Chernicoff

16 June 1995

Table of Contents

1 From Mac to Windows 9

Truth in Advertising....10
What This Book Is About....11
For More Information....12

2 System Foundations 15

Interface Files....15
Elementary Data Types....15
Hungarian Notation....17
Processes and Threads....19
Memory....21
How Virtual Memory Works....21
Heap Allocation....23
Memory-Mapped Files....25
Program Loading and Execution....26
Dynamic Link Libraries....27
Resources....28
Resource Identification....29
Resource Access....30
Defining Your Own Resources....32
The Same Thing...Only Different....40

3 The Windows Programming Model 45

Message Structure....46
Queued and Unqueued Messages....47
A Simple Windows Program....51
Main Function....51
Registering a Window Class....56

Creating and Displaying a Window 60
Main Loop....61
Window Procedures 63
Program Control Messages 66
Sending Messages to a Window 69
Message Operations 70
The Same Thing...Only Different 71

4 What's in a Window? 73

Anatomy of a Window....73
Child Windows....75
Pop-Up Windows....76
Z Order....77
Creating and Destroying Windows....77
Window and Class Attributes 80
Client Data....82
Window Display 84
Window-Related Messages....85
Other Window Operations 89
The Same Thing...Only Different 91

5 Mouse and Keyboard 93

The Mouse....93
Mouse Messages 95
Mouse Movements....97
Double Clicks....100
Capturing the Mouse....101
The Cursor....101
The Keyboard....104
Input Focus....104
Keystroke Messages....105
Character Messages....108
Polling the Keyboard 109
Dead Keys....109

Mouse and Keyboard Operations....111
The Same Thing...Only Different 112

6 The Graphics Device Interface 115

Graphical Data Structures 115
Points 115
Rectangles....116
Regions and Paths....117
Colors....119
Device Contexts....119
Device Context Attributes....120
GDI Objects 121
Obtaining a Device Context....123
Device Capabilities 125
Bitmaps and Memory Device Contexts 127
Clipping....128
Coordinate Systems....129
Mapping Modes 129
World Transforms....131
Line Drawing....132
Pens 132
Straight Lines....134
Curves 135
Shape Drawing....138
Brushes....138
Drawing Modes....141
Shapes 142
Low-Level Pixel Transfer 145
Metafiles 147
The Same Thing...Only Different 148

7 Controls 155

Notification and Control Messages....157
Keyboard Interface....159

Buttons 160
Button Messages 161
Pushbuttons 162
Checkboxes 163
Radio Buttons....163
Owner-Drawn Buttons 164
List Boxes....165
Creating a List Box 165
Adding and Retrieving List Items....166
Scrolling and Display....167
Owner-Drawn List Boxes 168
Selection Management....169
Scroll Bars....171
Static Controls....175
The Same Thing...Only Different 177

8 Text 181

Character Sets 181
The OEM Character Set....181
The ANSI Character Set 182
Character and String Operations....183
Edit Controls 184
Creating an Edit Control 185
Text Layout....187
Text Selection 189
Scrolling....190
Editing Text....190
The Caret....192
The Same Thing...Only Different 194

9 Menus 197

Menu Basics....197
Creating and Destroying Menus 200
Menu Messages....204

Keyboard Shortcuts....210
Mnemonics....210
Accelerators....212
Menu Operations....216
The Same Thing...Only Different....218

A WiniEdit Code Listing 221

Header File....221
Code File....225
Resource Header File....255
Resource Description File....257

B MiniEdit Code Listing 263

Make File....263
Header File....264
Code File....268
Resource Description File....307

Chapter 1

From Mac to Windows

Hi! Remember me? Back in 1984, I wrote a book called *Macintosh Revealed* that was the first widely available text on how to program Apple's Macintosh computer. In fact, for a time it was the *only* book available on Macintosh programming: Apple's own comprehensive manual on the subject, *Inside Macintosh,* didn't appear in a commercially published edition until nearly a year later. In the years since, *Macintosh Revealed* has gone through two editions, three publishers, and expanded from its original two volumes to a four-volume series. (I've lost count of how many volumes *Inside Macintosh* runs these days!) It's old and out of date by now, and left behind by the evolving Macintosh technology, but I'm amazed at how many people still come up to me, at trade shows and seminars, to tell me they got their start in Macintosh programming by reading my books. It's a great feeling to think that I may have helped the Macintosh grow into the great computer it is today.

All of us who were Macintosh devotees back in those early days knew that we were onto something revolutionary—a whole new way of interacting with computers. Our straitlaced friends with their stodgy old IBM clones could scoff at our cute little "toy" computer, but we knew better. Keyboards and command lines and 80-by-24 character screens were destined to become a thing of the past; the future belonged to mice and bitmaps, windows and menus, multiple fonts and character styles, pointing and clicking, cutting and pasting. Our clonehead friends were already obsolete; they just didn't know it yet.

Even when Microsoft introduced its Windows operating system for IBM-compatibles, we weren't impressed. Those early versions of Windows were sluggish, unresponsive, and limited by the relatively low resolution of the available screens. Because they were built on top of the cumbersome DOS operating system, they inherited all of its annoying limitations as well—such as a file system featuring interminable directory paths leading to local file names that couldn't be longer than eight characters with a three-character extension. Besides, the Windows user interface was such a transparent knock-off of the Macintosh. Why settle for a cheap imitation when you could have the real thing?

Well, a funny thing happened on the way to the future. Somewhere along the line, millions of users began flocking to Windows instead of the Macintosh. As the Windows user base grew, software developers began focusing more of their efforts on that platform and less on the Macintosh. Those of us who were diehard Macintosh loyalists still knew we had the best system going, but we could feel the tide turning against us. As time went on, it was becoming harder and harder to stay active and credible in the personal-computer industry without some level of Windows literacy.

So I had just about decided it was time to swallow my pride and try to learn a little about Windows programming when, by a fortuitous stroke of timing, my phone rang one day last fall. The caller introduced himself as James Plamondon from Microsoft, product evangelist for Windows 95, the new version of Windows due out Real Soon Now. James said he was familiar with my Macintosh books from his days as a Macintosh registered developer and was wondering if I'd be interested in doing a book on Windows programming for the Macintosh audience. I told him I was probably not the right person for the job: I was basically a Macintosh kind of guy who had made a career of staying as far as possible from any computer with three letters in its name or the name of its operating system. I'd written some documentation once for a client who insisted I do the work on a PC-compatible machine, but it had taken a powerful effort of will not to return the machine at the end of the contract with a brick through the screen. I hadn't laid eyes on Windows in years, and as far as DOS was concerned, I didn't know an **`AUTOEXEC.BAT`** from a cricket bat.

James waited patiently while I finished my tirade, then explained with equanimity that Windows had come a long way since I last saw it, that the new version had finally broken free of its dependence on DOS, that it was perfectly possible nowadays to run Windows without ever going near a **`CONFIG.SYS`**, and that for the purposes he had in mind, my almost total ignorance of anything to do with Windows was a Good Thing. He wanted someone with a Macintosh background, he said, to sit down and learn Windows for the first time and then share the experience with the Macintosh community. Well, here was as good an excuse as any to take the Windows plunge. James can be a very persuasive guy, but the fact is he didn't need to do much convincing to get me to do the book.

Truth in Advertising

In the interest of full disclosure, let's be clear about something right from the start: *Microsoft is paying me to write this book.* But I also want to be clear that they are exercising no editorial control over its content. I told James when I agreed to this project that I was not interested in writing Microsoft marketing literature. My deal with Microsoft was that I spend some time learning about the system and then write about it from a Macintosh point of view. I insisted on complete freedom to criticize as well as praise, to compare Windows to the Macintosh both positively and negatively. Any fault I might find with Windows is published here with Microsoft's full knowledge and consent. By the same token, anything I say in Windows' favor is there because *I really mean it*, not because Microsoft is paying the freight. The opinions expressed in this book are candid, unbiased, and entirely my own. Neither James nor I would have it any other way.

This book is what's known in publishing circles as a "work for hire." As the author, I get no royalties from sales of the book, just a flat fee for writing it. That's an important point, because it helps to ensure my editorial independence. Because I've already made every nickel I'm ever going to see from this project before the book ever hits the presses, I have no financial stake in its commercial success or that of Windows 95 itself. I don't care how many copies Microsoft sells or doesn't sell, either of the book or of the system. I have no incentive to try to sell you anything. I am *not* interested in persuading anyone to abandon the Macintosh platform and switch to Windows development. I'm certainly not about to do that myself; I love the Macintosh too much, and always will. I take it as given that if you're reading this book, it's because you've already decided, for reasons of your own, to learn how to write Windows programs. If that's not the case,

you have my permission to leave the book in the store and go spend the money on a nice dinner instead. No harm, no foul.

What This Book Is About

To get the most from this book, you should be

- an experienced Macintosh programmer
- familiar with the Macintosh Operating System and User Interface Toolbox
- acquainted with popular Mac development tools like Apple's Macintosh Programmer's Workshop, Symantec's Think C and Think Pascal, or Metrowerks' Code Warrior
- comfortable with the general concepts and principles of Macintosh-style event-driven programming

The purpose of the book is to show you how to put all that Macintosh knowledge to work in writing application software for the Windows platform. You'll be surprised (or not, depending on how you look at it) to find how much of your existing knowledge remains valid and useful on the Windows side of the fence. In each chapter, we'll consider one aspect of the Windows user interface or programming model—windows, menus, onscreen graphics, mouse and keyboard input, or whatever—and compare the Windows approach with what you already know about the subject from the Macintosh. Each chapter ends with a recurring section titled "The Same Thing...Only Different" that summarizes the similarities and differences between the two platforms.

The first thing I did to begin learning Windows myself was to take my old MiniEdit program, the simple cut-and-paste text editor I developed as a programming example for the *Macintosh Revealed* series, and port it to Windows. The conversion turned out to be remarkably easy. The fundamental structure of the program carried over almost intact from one platform to the other; the biggest difference between the two turned out to be that the Windows version—which I call WiniEdit—was substantially shorter than the Macintosh original. Part of the reason lies in the design of the Windows programming interface, which (as we'll learn later) implicitly handles of a lot of the implementation detail that Macintosh Toolbox programs have to do for themselves. But a large part of it, too, was the fact that the conceptual framework for Windows programming is so similar in many ways to that of the Macintosh.

Where applicable, I use code excerpts from WiniEdit to illustrate various points about Windows programming. It turns out, though—for many of the same reasons that WiniEdit is shorter than its Macintosh counterpart, MiniEdit—that it doesn't cover as much of the subject as MiniEdit does. To choose a couple of examples, MiniEdit had to include a substantial amount of code to support text editing and scrolling in its document windows. In Windows, much more of these mechanisms is implemented automatically by the system itself: there's simply a whole lot less code needed on the application side to support them. So we'll find, as we go along, that we'll spending less time examining the details of the program and more time discussing the subject on a theoretical level. If you really want to revel in all the glorious details, you'll find the complete code of the WiniEdit program in an appendix at the back of the book.

I should also put in a word here about programming languages. Back in the early days of the Macintosh, almost all programming for the machine was done in Pascal. That was the language in

which Apple documented all of the Toolbox interfaces in *Inside Macintosh*, and it was the natural choice for me to use in *Macintosh Revealed* as well. Times have changed since then, and today most programmers, on both the Macintosh and Windows, prefer C and its object-oriented extension, C++. So before porting MiniEdit to Windows, I first translated it from Pascal to C and then ported it from there. (Actually, although the program doesn't use any of the language's object-oriented features, I used C++ instead of straight C for the sake of a few syntactic conveniences, like function-style typecasting.) Since the only published version of MiniEdit is written in Pascal, I've included the C version in an appendix for convenient comparison.

For More Information

Let me stress that this book is intended as an *introduction* to Windows for Macintosh programmers. Its purpose is to give you an overview of Windows programming from the Macintosh point of view. I does *not* attempt to cover the subject in any great depth or detail, first because there's just too much ground to cover for a book this size, and second because I'm still just learning this stuff myself and I don't *know* all there is to say about it. Luckily, I can recommend a couple of other good books that you can use to put more flesh on the bare bones.

The definitive manual on all aspects of Windows programming is the *Win32 Programmer's Reference*, published by Microsoft Press. Where I might just mention the name of a function, message, or data structure in passing, the *Programmer's Reference* describes it comprehensively and exhaustively. It is *the* essential reference for all Windows programmers and an indispensable companion to the book you're reading now.

Most Windows programmers cut their eye teeth on *Programming Windows*, by Charles Petzold, also from Microsoft Press (or on one of its many variant editions—*Programming Windows 3.1*, *Programming Windows 95*, and so forth). Petzold has done for Windows programming what I tried to do for the Macintosh: a readable, informative introduction to the basic ideas and principles that every programmer needs to understand. It would be presumptuous of me to call Charles the Steve Chernicoff of Windows programming, but I would be honored to be considered the Charles Petzold of the Macintosh.

Another excellent source of information from Microsoft Press is *Advanced Win32 Programming*, by Jeffrey Richter. This is a more technical treatment of the lower-level recesses of the Windows system, such as processes, memory management, and the file system, in far greater depth than I could possibly have attempted here. I've found Richter's book indispensable in understanding these aspects of the system myself.

Finally, let me say that I welcome any and all comments, criticisms, errata, quibbles, or bug reports you might have on any aspect of this book. You can reach me by electronic mail at **`chernico@ccnet.com`**, or by snail mail in care of the publisher. I hope you find this book useful in making *your* move from Macintosh to Windows.

Chapter 2

System Foundations

Before we get to the good stuff, we have some dull but necessary business to take care of. In this chapter, we'll take a quick look at some of the basic foundations of the Windows system: elementary data types, multithreaded processes, memory management, program loading and execution, resources, and the file system. After we have these preliminaries out of the way, we'll be ready to talk about the really interesting stuff: programming the Windows user interface.

Interface Files

The C/C++ language version of the Win32 Application Program Interface (API) is defined by a set of header files that you incorporate into your program with a C **`#include`** directive:

```
#include <windows.h>
```

The master header file, **`WINDOWS.H`**, in turn calls in other header files, such as **`WINDEF.H`** (basic type definitions), **`WINBASE.H`** (system kernel), **`WINGDI.H`** (graphics device interface—like QuickDraw on the Macintosh), **`WINUSER.H`** (user interface structures and functions), and so on. Taken together, the complete set of header files defines all of the data structures and functions that constitute the Win32 API.

Elementary Data Types

The Win32 header files define a set of elementary data types and related constants to represent things like integers, booleans, characters, pointers, floating-point numbers, and so forth. These are generally derived from the built-in standard types of the C language, but spelled in full capitals instead of lowercase: **`INT`** and **`CHAR`**, for instance, instead of **`int`** and **`char`**. The Windows versions are carefully defined with conditional-compilation flags to maintain source-code consistency between different (16- and 32-bit) versions of the Windows system. For example, type **`INT`** represents a 16-bit integer or a 32-bit integer, depending on which version of the system it's compiled for. As a rule, you should use the Windows-defined data types in your programs in place of the built-in versions.

Table 2–1 shows the most important of the elementary data types. Most of them are self-explanatory, but the last, **`HANDLE`**, needs a bit of further elaboration. Unlike the Macintosh-style handles that you're familiar with, a Windows handle is simply an "opaque reference" to an underlying object. The handle's internal semantics are meaningful only to the Windows system itself; it has no inherent, straightforward meaning like a Macintosh "pointer to a pointer." What's

Table 2–1. Elementary data types

Type name	Meaning
VOID	Unspecified type
BYTE	Unsigned byte (8 bits)
WORD	Unsigned word (16 bits)
DWORD	Unsigned double word (32 bits)
BOOL	Boolean (**TRUE** or **FALSE**)
CHAR	Character
UCHAR	Unsigned character
INT	Signed integer
UINT	Unsigned integer
SHORT	Signed short integer (16 bits)
USHORT	Unsigned short integer (16 bits)
LONG	Signed long integer (32 bits)
ULONG	Unsigned long integer (32 bits)
FLOAT	Floating-point number
POINT	Point
RECT	Rectangle
HANDLE	Handle

actually inside the handle is none of your program's business: it might be a simple pointer, an indirect pointer, an index into an internal system table, or who-knows-what else.

This means that you can't "open up" a Windows handle and use it to put your hands directly on the underlying data structure. The only meaningful thing you can do with a handle is pass it to a Windows function and let the system manipulate the data structure for you. For instance, suppose **theWindow** is a handle to a window structure and you want to get the value of its **style** field, which contains a set of bit flags controlling the window's appearance on the screen. On the Macintosh, you would write something like this (ignoring for the moment the fact that Macintosh windows are actually referenced with simple pointers instead of handles):

```
windowStyle = (**theWindow).style;
```

That is, you'd dereference the handle twice to get at the underlying data structure and then select the structure's **style** field. You can't do this in Windows, because the handle is opaque: it isn't just a pointer to a pointer. Instead, you have to pass the window handle to a Windows function that retrieves and returns the value of the desired field. Windows provides all the functions you need to access the internal components of its data structures. (In this case, you'd use the **GetWindowLong** function, which we'll learn about in Chapter 3.)

The Windows API header files also define a series of utility macros for manipulating the elementary data types (see Table 2–2). You can pack two bytes together into a 16-bit word with **MAKEWORD** and get them back out again with **HIBYTE** and **LOBYTE**. Similarly, **MAKELONG** packs two words into a 32-bit long word, and **HIWORD** and **LOWORD** extract them.

Table 2–2. Utility macros for elementary data types

Macro name	Meaning
MAKEWORD	Pack bytes into word
HIBYTE	Extract high-order byte from word
LOBYTE	Extract low-order byte from word
MAKELONG	Pack words into long word
HIWORD	Extract high-order wordfrom long word
LOWORD	Extract low-order wordfrom long word

Many of the types shown in Table 2–1 also have corresponding pointer types, such as **PWORD** (“pointer to word”) and **LPWORD** (“long pointer to word”). In older versions of Windows, with their 16-bit addresses and segmented memory architecture, the pointer types beginning with **P** represented “near pointers” within the local memory segment, while those with **LP** stood for “far pointers” from one segment to another. As we’ll see, Win32 systems no longer have segmented memory or near and far pointers. In Win32, both the **P** and **LP** pointer types represent true, 32-bit virtual-memory addresses.

Hungarian Notation

The names of pointer types such as **PWORD** and **LPWORD** are examples of a style of naming convention that is followed throughout the Windows programming interface. In this naming style, the names of data types, variables, structure members, and function parameters commonly begin with a prefix of anywhere from one to four letters identifying the general type of data they represent, followed by the main descriptive part of the name. For example, whereas a handle to a menu record in the Macintosh Toolbox is called a **MenuHandle**, in Windows it’s an **HMENU** (“handle to menu”).

This style of naming is called *Hungarian notation*, and it was actually invented by a real, live Hungarian. His name is Charles Simonyi (or Símonyi Károly in proper Magyar), and he’s the [??? insert correct job title here ???] at Microsoft. Charles and I have known each other for a very long time, and I could tell you some amusing stories about him when he was just a teenager, newly arrived from Budapest, and the only words of English he knew were . . . but I digress. Charles is also one of the most remarkable, brilliant people I’ve ever met. He’s the only person I know, for instance, who can read Egyptian hieroglyphics. Unfortunately, his ideas about naming conventions can make even a simple C program read as if it were written in hieroglyphics, too.

Table 2–3. Common Hungarian prefixes

Prefix	Meaning
`by`	Byte (8 bits)
`w`	Word (16 bits, unsigned)
`dw`	Double word (32 bits, unsigned)
`i`	Integer (signed)
`n`	Short integer (16 bits, signed)
`l`	Long integer (32 bits, signed)
`c`	Count
`f`	Flag (boolean)
`ch`	Character
`s`	String (unterminated)
`sz`	String (zero-terminated)
`pt`	Point
`x`	Horizontal coordinate (16 bits, signed)
`y`	Vertical coordinate (16 bits, signed)
`rgb`	RGB color
`fn`	Function
`p`	Pointer
`lp`	Long pointer
`h`	Handle

Table 2–3 is your Rosetta Stone, showing some of the common Hungarian prefixes that you'll encounter in the Windows interface and what they mean. Actually, the Hungarian style is not entirely foreign to the Macintosh either. You can find traces of it in some parts of the Macintosh Toolbox (notably the Printing Manager) that were designed by programmers influenced by Charles's ideas.

I don't really mean to be too critical of those ideas, by the way—those remarks I made about Egyptian hieroglyphics were only for comic effect. Hungarian serves a very useful purpose in making an object's data type directly evident from its name. When you run across a function parameter named **`dwCursorID`**, for example, you know immediately that it's a double-word cursor ID; **`fVisible`** is a boolean visibility flag; and so on. If necessary, two or more of these prefixes can be combined, producing names such as **`lpszTitle`** ("long pointer to a string, zero-terminated, representing the title").

Hungarian-style names are used universally throughout the Windows programming interface, and anyone who wants to do Windows programming has to know how to read them. They're just not my style, though, and I don't use them in my own code. I prefer naming my variables in English.

Processes and Threads

Like Macintosh System 7 (or earlier versions of the Macintosh system under MultiFinder), Windows is a *multitasking* operating system. This means that a user can run more than one program at a time, switching between them by clicking in their windows on the screen. The system keeps track of each program's state of execution and switches the processor's attention from one to another as needed.

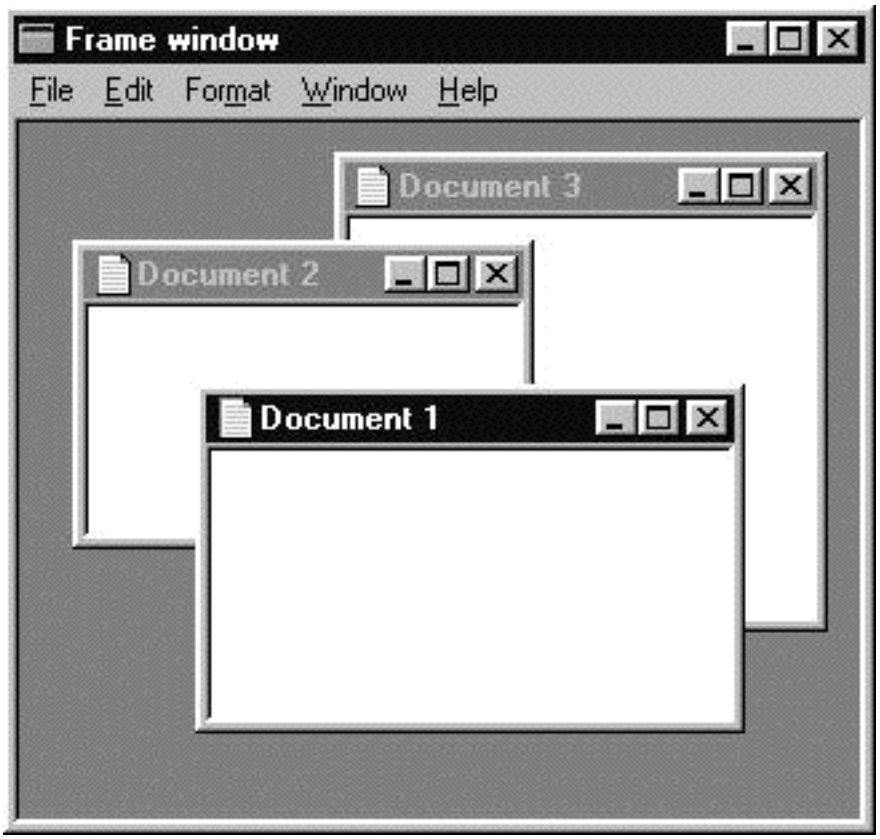
The Windows approach to multitasking is different from that of the Macintosh, however. The Macintosh uses a *cooperative multitasking* model, which relies on each program to be a "good citizen" and voluntarily give up control of the processor when it isn't needed. By using the Toolbox routine **`WaitNextEvent`** to retrieve events from its event queue, a program gives the system permission to suspend its execution and give control of the processor to some other program in its place. Eventually, the system will take advantage of one of the second program's **`WaitNextEvent`** calls to switch control back to the first. Everything works smoothly as long as every program calls **`WaitNextEvent`** often enough to give the others their turn at bat.

Windows, by contrast, uses a *preemptive multitasking* approach. The system itself maintains control of the processor and portions it out among the running programs in discrete "time slices." Each program behaves as if it had the computer all to itself. Programs don't have to do anything explicit to relinquish control; the system can forcibly yank the processor away from one program and turn it over to another.

Also unlike the Macintosh, Windows allows the user to run more than one *instance* of the same program simultaneously. Each separate instance of the program runs in its own *process*, independent of any others. There can be as many instances of the same program as the user wishes to start. All instances of the same program share the same copy of the program's executable code, but each has its own private set of data to work with.

This ability to create multiple instances of the same program can change the way you approach an application. My old MiniEdit program for the Macintosh, for instance, is a multiple-window text editor, allowing the user to view several text files side by side on the screen and to cut and paste text from one to another. When I ported the program to Windows, I realized that multiple windows were unnecessary. The Windows version of the program, WiniEdit, opens just one window on the screen, viewing a single file at a time. When the user asks to open a new file, it replaces the old file in the program's one window instead of opening in a separate window of its own. This is no limitation, though: a user who wants to view two or more files simultaneously can simply start up a new instance of the program for each file. Cutting and pasting from one file to another is no problem, since all such operations go through a global clipboard that's shared by all running processes. Restricting the program to just one window and one file at a time helps simplify the code and is one reason (among others) why the Windows version of the program is significantly shorter than its Macintosh counterpart.

Figure 2–1. An MDI frame window



Programs wishing to display multiple files on the screen simultaneously can use the Win32 Multiple Document Interface (MDI). An MDI program displays a single *frame window*, which in turn hosts a variable number of subwindows ("child windows," in Windows terminology), each containing a different document. The user can open and close document subwindows as needed and can freely move, size, and arrange them within the confines of the enclosing frame window (see Figure 2–1). Many common Windows applications use the MDI model, including the Windows NT Program Manager and File Manager, Microsoft Word and Excel, and Visual C++. The Win32 API provides all the support needed to write an MDI application; see the *Win32 Programmer's Reference* for details. However, MDI also introduces a significant extra level of programming complexity. Before deciding on an MDI implementation, consider seriously whether your user might not be just as well served by multiple instances of a single-document application instead.

Windows goes the Macintosh one step further in multitasking by allowing a program to spawn multiple threads of execution. A *thread* is an independent locus of control that shares processor time with all of the other threads in the system. (In a multiprocessor system, two or more threads might even run simultaneously on separate processors.) Your program might do background printing, for instance, by spawning a separate thread to print a document while the main thread continues to receive and process mouse and keyboard input from the user. Or you might

implement your own memory allocation scheme with a background garbage collector running in a thread of its own. All of a program's threads operate within the context of a single process and share access to the same address space and other system resources. To prevent threads from interfering with each other by independently accessing the same memory or resources, Windows offers a variety of facilities for synchronizing thread operations, such as wait functions, semaphores, and mutexes (short for "*mut*ually *ex*clusive access"). We won't be discussing multithread programming and synchronization in this book, but you can learn about them in the *Win32 Programmer's Reference* if your application calls for them.

Memory

Memory has had a long and checkered history in the Intel world. Even if you've never programmed an Intel-based machine, you've probably heard horror stories from your beleaguered DOS friends about segment registers, expanded memory, extended memory, protected mode, real mode, near pointers, far pointers, huge pointers, and the infamous "640K barrier." The early versions of Windows, which functioned within the DOS environment and relied on it for their memory management, inherited all of these headaches as well.

The good news is that a new day has dawned for Windows memory management. If you're an old hand at Intel/DOS/Windows programming, you can forget everything you've ever learned about memory. (Then again, if you're such an old hand at it, why are you reading this book?) If you're a newcomer like me, you don't have to know anything about the bad old days except that they're gone. In Win32, every process (that is, every instance of every program) gets its very own, totally linear, transparently paged, 4-gigabyte virtual address space. No segments. No modes. No barriers. A pointer is a pointer is a pointer.

Of course, the operative word in the preceding paragraph is "virtual." They haven't built a personal computer yet with 4 gigabytes of RAM, and if they did, you couldn't afford to buy it. (Anyway, I know *I* couldn't!) What really happens is that the Windows operating system, with hardware support from the Intel processor (80386 or later), gives the program the *illusion* of a 4-gigabyte playpen to throw its toys around in. In practice, a program will actually use only a fraction of this vast address space—and typically, only a fraction of *that* will be physically present in RAM at any one time.

How Virtual Memory Works

On the Macintosh, virtual memory is a luxury available only on high-end models—those equipped with a paged memory-management unit, or PMMU. Even on those systems, it's strictly optional: the user chooses whether to activate it or not (using the Memory control panel), and the system can function perfectly well without it. In Win32, virtual memory is a necessity, woven inextricably into the fabric of the system itself.

As in any virtual memory system, the Windows operating system is responsible for maintaining the correspondence between logical addresses seen by the program and physical storage locations in RAM or on the disk. To simplify the system's internal bookkeeping and reduce memory fragmentation, both the logical address space and the physical storage media are divided into units of uniform size called *pages*. A typical page size is 4K (4096) bytes, but this can vary from

one hardware platform to another: the DEC Alpha implementation of Windows, for instances, uses an 8K page.

The essence of virtual memory is that not all of a program's logical address space has to be physically present in RAM at all times. The system creates a *paging file* on the disk to hold pages that are not currently in RAM, and a *page map* to keep track of each page's current location, either in RAM or on the disk. When a program attempts to refer to a logical address on a page that is not currently in RAM, a hardware *page fault* occurs. The system then locates the required page on the disk and reads it into RAM, replacing one of the pages already there. The page being replaced may belong to the same process (that is, the same instance of the same program) as the page being read in, or to some other process. If the contents of the page are "dirty" (have been changed while in RAM), they are copied back out to the disk before being overwritten. This page swap is transparent to the running program, which has the illusion of a 4-gigabyte linear address space all to itself.

Establishing the correspondence between logical addresses and physical storage locations is a two-step process. First the program *reserves* a range of addresses in its address space, known as a *region.* Reserving a region doesn't actually allocate any physical memory resources yet; it simply informs the system that the program intends to use the logical addresses within the designated range. When the program no longer needs this block of addresses, it *releases* the region to make it available for reuse.

A reserved region is always a whole number of pages in size (the system will adjust it upward, if necessary, to the nearest multiple of the page size) and always begins at an address that is a multiple of the system's *allocation granularity.* Like the page size, the allocation granularity can vary from one platform to another, but it is typically 64K (65,536) bytes. A program can use the Windows function **`GetSystemInfo`** to learn the page size and allocation granularity, as well as a variety of other parameters of the hardware platform it's running on.

Once a region of address space has been reserved, physical storage resources must be *committed* to it before it can be used. Once again, the unit of storage commitment is the page. Committing a page of address space gives it a physical location on the disk in which to hold its contents when they're not active in RAM. In committing a page, a program can specify its level of access protection: possible protection attributes include familiar ones such as read-only, read/write, execute-only, and so on, as well as a few other, more specialized options that we'll be discussing shortly. The program can commit all of the pages in a region at once, in a single operation, but this isn't required: they can also be committed a page at a time, as needed. When the physical storage space is no longer needed, the page can be *decommitted.*

Reserving a region of address space and committing physical storage to it can be done separately or combined in a single operation. The Windows function **`VirtualAlloc`** performs either or both operations, depending on the values of the parameters passed to it. Similarly, the **`VirtualFree`** function can decommit physical storage, release a logical region, or both.

Heap Allocation

What "memory management" means to most Macintosh programmers is allocating and freeing individual blocks of memory from a heap. Needless to say, Windows has heaps too. They don't

work identically to those on the Macintosh, but there's nothing particularly mysterious about them, either.

In the older 16-bit versions of Windows, each program had its own *local heap,* along with a system-wide *global heap* that was shared in common by all programs (sort of like the Macintosh application and system heaps). A program could create blocks in either heap with the Windows functions **`LocalAlloc`** and **`GlobalAlloc`**, and destroy them with **`LocalFree`** and **`GlobalFree`**. When creating a block, you could specify whether it should be *movable* or *fixed* (analogous to Macintosh relocatable or nonrelocatable blocks) and *discardable* or *nondiscardable* (like purgeable or nonpurgeable on the Macintosh).

These old allocation functions would return a handle to the new block, but remember that a Windows handle is merely an "opaque reference" to an object, whose internal contents are meaningful only to the system. In the case of the old heap allocation functions, the handle returned was actually a 16-bit index to an entry in a descriptor table, which in turn held the address of the block along with other information about it. To gain access to the block's contents, you had to call the Windows function **`LocalLock`** or **`GlobalLock`** (depending on which heap the block was in). These functions would lock the block in place, preventing it from being moved or discarded, and return a pointer that you could use to get at its contents. There was literally no way to touch the block without locking it at the same time. When you were finished manipulating the block, you would unlock it again with **`LocalUnlock`** or **`GlobalUnlock`**, after which your pointer to it was no longer valid:

```
blockHandle = LocalAlloc(...);          // Allocate block from local heap
. . . ;
blockPtr = LocalLock (blockHandle);     // Lock block and get pointer
   . . . *blockPtr . . . ;              // Use pointer to access block
LocalUnlock (blockHandle);              // Unlock block; pointer no longer valid
```

Handles still exist in Win32, but only for objects that the system creates on your behalf, such as windows or menus. (Needless to say, they're also 32 bits long now instead of 16.) The old **`LocalAlloc`** and **`GlobalAlloc`** and their related functions have been replaced with a new series called **`HeapAlloc`**, **`HeapFree`**, and so on. The old calls are still supported, but only for the sake of backward compatibility with existing code. New programs being developed today are supposed to use the new functions exclusively.

The biggest difference between the old and new heap functions is that with the new ones there are no movable or discardable blocks and no heap compaction. (There is a function called **`HeapCompact`**, but all it does is coalesce adjacent free blocks and decommit empty pages; it never moves or discards any existing blocks.) I guess the theory is that with 4 gigabytes of address space available, there's always room to grow the heap if you have to, so compaction is unnecessary. Whatever the reason, the fact is that every block in a Win32 heap is fixed and nondiscardable, and is referenced with a direct pointer instead of a handle.

Table 2–4. Win32 heap allocation functions

Function	Mac counterpart	Purpose
`HeapCreate`	`InitZone`	Create heap
`HeapDestroy`	————	Destroy heap
`GetProcessHeap`	`ApplicationZone`	Get program's default heap
`HeapAlloc`	`NewHandle`, `NewPtr`	Allocate block from heap
`HeapFree`	`DisposHandle`, `DisposPtr`	Deallocate block from heap
`HeapSize`	`GetPtrSize`	Get size of block in heap
`HeapReAlloc`	`SetPtrSize`	Change size of block in heap

Table 2–4 shows the basic functions in the new Win32 series. **`HeapAlloc`** creates a new block of a specified size and returns a pointer (not a handle!) to it. **`HeapFree`** deallocates an existing block; all pointers to the block become invalid and must not be used again. **`HeapSize`** returns the size of an existing block in bytes; **`HeapReAlloc`** changes the size of an existing block.

> The **`HeapReAlloc`** function can always shorten a block in place, but if you ask to lengthen the block, it may need to be moved to another location within the heap in order to find the needed space. Instead of updating a master pointer as on the Macintosh, **`HeapReAlloc`** simply returns a pointer to the block's new location; it's up to you to use the new pointer in place of any previous ones you may have been holding to the block. Of course, this can lead to trouble if you have lots of copies of the old pointer sitting around embedded in other data structures. If you want to reallocate a block to a bigger size, you'd better make sure you have a way of tracking down and updating all of the existing pointers to it.

A little-used feature of the Macintosh Toolbox is the ability to maintain multiple heaps (called "heap zones" on the Macintosh). All Macintosh memory allocation routines operate implicitly on the "current heap zone." This is normally the application heap, but you can switch to another zone with the Toolbox routine **`SetZone`**. Win32 allows you the same capability, but instead of using a "current heap," each of the new heap allocation functions expects you to tell it explicitly what heap to operate on by supplying a handle to the heap as one of the function's parameters.

Usually, you'll just want to use your *process heap* for all of your memory allocation. Every instance of a Windows program has its own process heap, created automatically by the system when the instance is started up. Unless you're doing something fancy, you can simply obtain a handle to your process heap from the Win32 function **`GetProcessHeap`** and pass it as a parameter whenever you call one of the heap functions. Sometimes, though, you may find it useful to create additional heaps with the **`HeapCreate`** function. For instance, if your program creates and destroys lots of memory blocks of a certain uniform size (nodes in a linked list or tree structure, perhaps), you can manage memory more efficiently by giving them a heap of their own. Since all blocks in this heap are the same size, it can never become fragmented: as long as there's any free space at all, it's guaranteed to be the right size for the block you want to create. For a few other ways to use multiple heaps, see Jeffrey Richter's book, *Advanced Win32 Programming.*

To create a new heap zone on the Macintosh, you allocate a large nonrelocatable block from within your existing application heap (or even, in rare cases, from the stack) and pass it to the Toolbox function **`InitZone`** to give it the internal data structures it needs to function as an independent heap. In effect, the new zone becomes a "subheap" inside your main application heap. In Win32, the **`HeapCreate`** function creates a separate new heap in its own region of address space, rather than inside another existing heap. You specify the new heap's initial and maximum size; **`HeapCreate`** does the following:

- Reserves the specified maximum number of bytes in your address space
- Commits the specified initial amount of physical disk space
- Gives you back a handle to the new heap, which you can then pass to any of the other heap allocation functions

If your subsequent allocation requests exceed the initial size of the heap, the system will automatically commit additional pages of physical storage as needed, up to the reserved maximum. When you're finished using the new heap, you can decommit the storage it occupies with **`HeapDestroy`**. If you don't explicitly destroy the heap, its storage is automatically decommitted when your program terminates.

Memory-Mapped Files

One of the side benefits of the way Windows virtual memory works is a new way of doing file input and output. To read or write a file on the Macintosh, you have to allocate an input/output buffer in your heap and then explicitly transfer data between the buffer and the file, using either the high-level input/output routines **`FSRead`** and **`FSWrite`** or their low-level, parameter-block-based counterparts **`PBRead`** and **`PBWrite`**. You can do your I/O the same way in Windows, using the file transfer functions **`ReadFile`** and **`WriteFile`**—my WiniEdit example program does it this way, for instance—but often it's more convenient to use a *memory-mapped file* instead.

The idea of memory-mapped files follows from the way the Windows virtual memory system is structured. Although the disk space backing up a page of memory normally comes from the system's paging file, an entry in the page map can actually point anywhere at all on the disk. In particular, it can point to a disk page that's part of an ordinary data file. This makes it possible to access the contents of a file by simply mapping them directly into a region of your program's virtual address space. You can then fetch or store into any byte of the file just as if it were an ordinary memory location—because, in effect, that's just what it is. The system transparently handles all the details of physically transferring the data in and out to the disk.

The first step in memory-mapping a file is to create a *file mapping,* using a Win32 function named—guess what?—**`CreateFileMapping`**. The mapping is a system-created object that identifies the file along with some other security and protection attributes to control the type of access allowed. Once you have a file mapping, you can use it to open a *view* of all or part of the file within your virtual address space. The **`MapViewOfFile`** function returns a pointer to the region of address space containing the file; you can then access the file's contents by indexing from this pointer:

```
fileHandle = CreateFile (...);                    // Open the file
fileMap    = CreateFileMapping (fileHandle,...);  // Create mapping
fileBase   = MapViewOfFile (fileMap,...);         // Open a view

. . . fileBase[i] . . . ;                         // Access file as an array
. . . *(fileBase + i) . . . ;                     //    or by pointer arithmetic

UnmapViewOfFile (fileBase);                       // Close view when finished
CloseHandle (fileMap);                            // Destroy mapping
```

Memory-mapped files are useful in their own right, just for the convenient access they offer to the contents of a file. In addition, they provide a way for separate processes or threads to communicate with one another, by sharing information through independent views of the same file. This is perfectly safe as long as only one of the processes can write to the file, with the rest restricted to read-only access. It can get tricky, though, if two or more processes try to write to the file independently. One necessary precaution is to make sure all of the processes' views of the file are based on the same mapping object. This ensures that the views remain *coherent*, meaning that any changes made via one view will be seen by the others as well. In addition, the processes must use some sort of synchronization mechanism, such as a mutex, to ensure that only one process at a time has permission to write to the file. Unfortunately, this whole topic of interprocess synchronization is beyond the scope of this book; see the *Win32 Programmer's Reference* or the Jeffrey Richter book, *Advanced Win32 Programming*, for more details.

Program Loading and Execution

Now comes the really cool part. If any file on the disk can be mapped into a program's virtual address space, how about the file containing the executable code of the program itself? In fact, this is just the way the Windows system loads a program into memory for execution: it simply opens a memory-mapped view of the code from the program's executable file (which normally carries the file extension **`.exe`**, a vestige of the old DOS file system) within the program's virtual address space.

Opening a view of the executable file does not, in itself, cause any code to be loaded from the disk; but when the system attempts to transfer control to the program's main entry point, a page fault will occur that will load in the first page of code and begin executing. Thereafter, any time the program transfers control to an address that is not already in RAM, the virtual memory system will bring in the missing page automatically. There's no need to break the code up into explicitly defined segments, as on the Macintosh; code swapping takes place transparently as part of the routine workings of virtual memory.

Notice that this scheme makes it easy for multiple instances of a program to share the same copy of the code: the system simply maps the same executable file into each instance's address space, allowing them to page in portions of the code as needed. If two instances happen to be executing in the same page of code, they can share the same physical copy of that page in RAM.

Of course, each instance has to have its own private region of address space to hold its copy of the program's global variables. The way Windows manages this is also instructive. Space for the global variables is reserved within the program's executable file. When these pages are mapped into a process's address space, they are given a special protection attribute called *copy-on-write*.

This allows the process to read the page's contents freely, so long as it doesn't try to modify them. The first time the process attempts to write to such a page, the system automatically creates a new copy of the page in the system paging file and maps the new copy into the process's address space in place of the original copy from the program's executable file. Multiple instances of the same program can thus share the identical copy of the data page as long as they're only reading from it, while getting their own private copies to work with if they have to write into it.

Dynamic Link Libraries

One of the key early design decisions on the Macintosh was to have the executable code of the Toolbox reside permanently in ROM. Calls from a running application program to a Toolbox routine are implemented via the Motorola processor's trap mechanism. Each Toolbox call generates a *trap word* that looks just like an ordinary machine-language instruction, but actually triggers the processor's *unimplemented instruction* trap. This activates the Macintosh system's Trap Dispatcher, which decodes the trap word to determine what Toolbox routine it refers to, locates the routine's entry point in ROM, and transfers control to it. (All this is different, of course, on the new-fangled Power Macs.)

As you might expect, Windows system calls work differently. The Intel-based machines that Windows typically runs on have no built-in ROM code and no processor emulation mechanism similar to the Motorola unimplemented instruction trap. All of the code for the Windows system resides on the hard disk in a collection of *dynamic link libraries*, or *DLLs*. DLLs are as fundamental to the operation of Windows as the trap mechanism is to the Macintosh Toolbox.

A dynamic link library is simply a collection of machine-language routines that are available to be called by any program that needs them. There are many DLLs that make up the Windows system, but the most important of them are **`KERNEL.DLL`**, which contains the code for low-level system functions such as memory and process management; **`GDI.DLL`**, which contains the Graphics Device Interface (general-purpose graphics routines); and **`USER.DLL`**, which implements the high-level Windows user interface. Notice that these three libraries correspond to the three main divisions of the Macintosh ROM code: the Macintosh Operating System, QuickDraw, and the User Interface Toolbox.

The reason they're called *dynamic* link libraries is that the program's references to routines in the library are resolved dynamically, when the program is loaded and run, rather than statically, at build time. Instead of linking the library code directly into the program's executable file, the Windows linker simply compiles a table of system routines the program calls and the DLLs in which they reside. When the program is run, the system locates the needed DLLs on the disk, maps them into the program's virtual address space, and patches each table entry to the correct virtual address. This allows multiple running programs to share the same copy of the system code, leaving it to the virtual memory system to page the code in from the disk only when it's actually needed.

Resources

I've always considered resources the key to Macintosh programming. Although they were first invented to serve a very limited purpose—translating onscreen messages and menu commands for

use in foreign countries—they turned out to be such a useful device that they have grown to take over the entire Macintosh software architecture. Many of the familiar software conveniences that Macintosh users know and love—desk accessories, **INIT** extensions, control panels, Chooser devices—are made possible by resources. I suspect that if Apple were to redesign the Macintosh Toolbox from scratch today, they would scrap the idea of a file's data fork entirely and just make everything a resource and every file a resource file.

Windows has resources too, but they're not as pervasive a part of the overall software architecture as they are on the Macintosh. Unlike the Macintosh, with its hundreds of resource types serving every conceivable purpose, Windows has only about a dozen built-in resource types, shown in Table 2–5. Their only purpose is to isolate certain aspects of a program's operation so that they can be easily modified without affecting the code itself.

Most of the resource types shown in the table are self-explanatory, but a few of them bear further discussion. As we'll learn in Chapter 7, Windows uses the term *menu* to refer, not only to the individual lists of items that we think of as menus on the Macintosh, but also to what the Macintosh calls a *menu bar:* the complete set of individual menus that a program offers to the user. So the Windows **RT_MENU** resource is actually a combination of the Macintosh resource types **'MBAR'** and **'MENU'**. (The prefix **RT_** stands for "resource type.") Also, a Macintosh **'MENU'** resource includes information about Command-key shortcuts that the user can use to invoke menu items directly from the keyboard. In Windows, these shortcuts are called *keyboard accelerators* and are defined in a separate *accelerator table* obtained from a resource of type **RT_ACCELERATOR**. The Windows resource type **RT_STRING** is not a single text string like a Macintosh **'STR '** resource, but rather a *string table*, like **'STR#'** on the Macintosh. (Individual strings within the table have their own resource IDs, however, rather than a single ID for the whole table and a separate index for each string, as on the Macintosh.) Finally, the **RT_GROUP_CURSOR** and **RT_GROUP_ICON** resources are bundles of alternative versions of a cursor or icon, for use on display screens of different depths and resolutions. When your program asks to load the cursor or icon, Windows checks the properties of the current display device on the user's machine and automatically chooses the version most suitable for that device.

Table 2–5. Standard resource types

Type name	Access function	Meaning
`RT_MENU`	`LoadMenu`	Menu
`RT_ACCELERATOR`	`LoadAccelerators`	Accelerator table
`RT_DIALOG`	————	Dialog box
`RT_FONT`	————	Font
`RT_FONTDIR`	————	Font directory
`RT_CURSOR`	`LoadCursor`	Cursor
`RT_GROUP_CURSOR`	————	Cursor
`RT_ICON`	`LoadIcon`	Icon
`RT_GROUP_ICON`	————	Icon
`RT_STRING`	`LoadString`	String table
`RT_BITMAP`	`LoadBitmap`	Bitmap
`RT_MESSAGETABLE`	`FormatMessage`	Message table entry
`RT_VERSION`	————	Version data
`RT_RCDATA`	————	Application-defined resource

Resource Identification

Since Windows files don't have separate resource forks, resources have to reside in the main body of a file (what the Macintosh would call its data fork). In the context of resource management, a file containing resources is called a *resource module.* Three types of file can serve as resource modules:

- Executable program files (file extension `.exe`)
- Dynamic link libraries (file extension `.dll`)
- Font files (file extension `.fon`)

A program's own private resources typically reside in its executable file. There are also some system-wide standard resources that live in the Windows DLLs. Windows has no equivalent to the Macintosh notion of a "current resource file": all Windows functions for retrieving resources accept an explicit parameter to identify the module in which to look for the desired resource. Typically, this is a handle to a program instance, designating the program's executable file as the resource module; a `NULL` value denotes one of the standard system-defined resources instead.

As on the Macintosh, a Windows resource is designated by its resource type and an individual identifier within that type. In Windows, each of these two items can be given as either a character string or an integer (somewhat like identifying a Macintosh resource either by name or by ID number). Although either method of resource identification is acceptable, integer identifiers are considered preferable to strings because they use less memory space. Formally, all Windows functions that operate on resources take string parameters for both the resource type and the individual ID; but as we'll see, there are a variety of ways to pass them the identifiers in integer form instead.

Resource Access

In the general case, accessing a resource is a two-stage process: first you have to locate the resource within its resource module, then load its contents into memory. The general-purpose Windows functions for these two operations are **FindResource** and **LoadResource**. **FindResource** takes three parameters, specifying the resource's module, identifier, and type. It returns a handle, not to the resource itself, but to the information block describing it in the resource module. (If the requested resource can't be found, the function returns **NULL**.) You must then pass this handle, in turn, to **LoadResource** (along with the parameter identifying the resource module again) to get a handle to the data of the resource itself. For example the following statements find and load the system's standard arrow cursor:

```
rsrcHandle = FindResource(NULL, "IDC_ARROW", "RT_CURSOR");  // Find resource info
datahandle = LoadResource(NULL, rsrcHandle);                // Load resource data
```

(The string **"IDC_ARROW"** is the name of the resource containing the standard arrow cursor; the prefix **IDC_** stands for "identifier of a cursor.")

In practice, however, it usually isn't necessary to perform these two separate steps explicitly. Many of the standard resource types have their own special-purpose access functions (listed in Table 2–5), which combine the two steps into a single operation. To load the standard arrow cursor, for instance, you'll normally use the **LoadCursor** function rather than **FindResource** and **LoadResource**:

```
arrowHandle = LoadCursor(NULL, "IDC_ARROW");
```

This function already knows what resource type to look for, so you don't have to spell it out with a parameter. It performs the operations of both **FindResource** and **LoadResource** in a single call, and gives you back a specifically-typed handle of type **HCURSOR**. The other functions listed in the table work similarly: **LoadMenu** returns a handle of type **HMENU**, **LoadIcon** a handle of type **HICON**, and so on.

Although the resource ID parameter to each of these functions is nominally defined as a string (that is, as a pointer to a character), you can actually pass an integer value instead. If the high-order word of the 32-bit parameter is zero, the resource-access functions will recognize it as an integer resource identifier instead of a string pointer. If you happened to know, for instance, that the numerical identifier for the **IDC_ARROW** cursor is **32512**, you could pass that value as the second parameter to **LoadCursor**. Of course, you would first have to pad the 16-bit integer value to 32 bits and typecast it into a string pointer. As a convenience, the Win32 interface defines a C macro named **MAKEINTRESOURCE** that does just that:

```
#define MAKEINTRESOURCE(i) (LPSTR)((DWORD)((WORD)(i)))
```

(The actual definition is a bit more complex than this, but conceptually this is the way it works.) So to load the arrow cursor, you could use the following code:

```
rsrcID      = MAKEINTRESOURCE(32512);
arrowHandle = LoadCursor(NULL, rsrcID);
```

If the first character of the parameter string is a number sign (**#**), the resource-access functions interpret the string as a decimal integer representing the resource ID—so you could also load the arrow cursor this way:

```
arrowHandle = LoadCursor(NULL, "#32512");
```

Of course, it isn't very good coding style to hard-wire that resource number directly into your code. The Win32 headers define the names of all the standard resources as constants, already padded and typecast with **MAKEINTRESOURCE a**nd ready to use:

```
#define IDC_ARROW  MAKEINTRESOURCE(32512)
```

So instead of using the integer value directly, you can just use the resource name as a parameter to the resource-access function:

```
arrowHandle = LoadCursor(NULL, IDC_ARROW);
```

Notice that this is not the same as the first example we looked at: there are no string quotes around the name this time. In the first example, the parameter we passed was a string containing the name of the desired resource; here, it's a constant representing the integer resource ID, padded to 32 bits and typecast to an appropriate string pointer.

When you're working with the standard resource types, you usually aren't interested in opening up the resource and looking at the data inside; you simply get a handle to the resource and pass it off to some other Windows function to operate on. If you define your own custom resource types, however, only you know what's inside them—so the only way to make use of them is to reach in and put your hands directly on the internal data. To do this, you have to convert the opaque handle you receive from **LoadResource** into an actual pointer to the beginning of the resource's data. The Windows function that does this conversion for you is **LockResource**. In older versions of the Windows system, **LockResource** did what its name implies: it locked the resource in place in the heap, so that it couldn't slide out from under your pointer in the event of a heap compaction. As we've seen, the Win32 memory manager no longer moves blocks around in the heap; but you still use the **LockResource** function to get a pointer to a resource's data:

```
dataPtr = LockResource(dataHandle);       // Convert handle to pointer
. . . dataPtr[i] . . . ;                  // Access data as an array
. . . *(dataPtr + i) . . . ;              //    or by pointer arithmetic
UnlockResource(dataHandle);               // Unlock when finished
```

The closing call to **UnlockResource** is no longer needed (and in fact has no effect at all in Win32), but you can still retain it for the sake of symmetry and good form.

Defining Your Own Resources

Of course, many if not most Windows programs will need to define some resources of their own, in addition to the standard ones built into the Windows system. The Visual C++ development system includes a resource compiler similar to Rez in the Macintosh Programmer's Workshop. The resource compiler accepts a text file (file extension **.rc**, for "resource compiler") containing a description of the program's resources and compiles the resources themselves into the program's executable file. For illustration, Listing 2–1 shows the contents of **WiniEdit.rc**, the resource description file for my WiniEdit example application program. In all, the file defines six resources: an icon for representing the program on the screen, a menu, an accelerator table (which defines keyboard shortcuts for the program's menu commands), a dialog box for the **About WiniEdit...** command, and two string tables containing various bits and pieces of text that the program uses.

Luckily, it isn't necessary to learn the syntax of the resource description language and compose your own resource descriptions in text form. The Visual C++ software development environment includes a set of interactive resource editors that allow you to create and manipulate resources

Listing 2–1. WiniEdit resource description file

```
//Microsoft Visual C++ generated resource script.
//
#include "WiniEdit Resources.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

#ifdef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "WiniEdit Resources.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

////////////////////////////////////////////////////////////////////////////
#endif    // APSTUDIO_INVOKED
```

directly onscreen with your mouse and keyboard, much like the Macintosh utility program ResEdit. There's a resource editor for menus, one for dialog boxes, and so on. The resource editors are fully integrated into the development environment. As you build your resources onscreen, the editors automatically generate the corresponding text descriptions and place them in the resource description file for you. Later, when you build your program, your Visual C++ make file (also generated for you automatically) will call the resource compiler to compile the resource descriptions into resources and place them in your executable file. In the normal course of events, you never have to deal directly with the contents of the resource description file yourself. All of the text in Listing 2–1 was generated for me by the Visual C++ development software, untouched by human hands. I don't fully understand everything that's in this file, and that's just fine with me.

Listing 2–1. WiniEdit resource description file (*continued*)

```
////////////////////////////////////////////////////////////////////////
//
// Dialog
//

About_Dialog DIALOG DISCARDABLE  32, 32, 163, 96
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE
FONT 8, "MS Sans Serif"
BEGIN
    ICON            ProgIcon_ID,About_Dialog,8,8,18,20
    CTEXT           "WiniEdit 1.0",IDC_STATIC,61,12,41,8
    CTEXT           "Example Windows application",IDC_STATIC,32,32,98,8
    LTEXT           "S. Chernicoff",IDC_STATIC,8,52,44,8
    RTEXT           "15 January 1995",IDC_STATIC,100,52,55,8
    DEFPUSHBUTTON   "Continue",IDOK,60,68,43,20
END

////////////////////////////////////////////////////////////////////////
//
// Menu
//

Main_Menu MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New\tCtrl+N",                New_Item
        MENUITEM "&Open...\tCtrl+O",            Open_Item
        MENUITEM "&Close\tCtrl+W",              Close_Item
        MENUITEM SEPARATOR
        MENUITEM "&Save\tCtrl+S",               Save_Item
        MENUITEM "Save &As...\tCtrl+Alt+S",     SaveAs_Item
        MENUITEM "&Revert to Saved...\tCtrl+R", Revert_Item
        MENUITEM SEPARATOR
        MENUITEM "Page Set&up...\tCtrl+Alt+P",  Setup_Item
        MENUITEM "&Print...\tCtrl+P",           Print_Item
        MENUITEM SEPARATOR
        MENUITEM "E&xit\tCtrl+Q",               Exit_Item
    END
    POPUP "&Edit"
    BEGIN
        MENUITEM "&Undo\tCtrl+Z",               Undo_Item
        MENUITEM SEPARATOR
        MENUITEM "Cu&t\tCtrl+X",                Cut_Item
        MENUITEM "&Copy\tCtrl+C",               Copy_Item
        MENUITEM "&Paste\tCtrl+V",              Paste_Item
        MENUITEM "&Delete\tDelete",             Delete_Item
        MENUITEM SEPARATOR
        MENUITEM "Select &All\tCtrl+A",         SelectAll_Item
    END
```

Listing 2–1. WiniEdit resource description file (*continued*)

```
    POPUP "&Format"
    BEGIN
        MENUITEM "Text &Format...\tCtrl+F",     Format_Item
        MENUITEM "&Default Format\tCtrl+D",     Default_Item
        MENUITEM SEPARATOR
        MENUITEM "&Background Color...\tCtrl+B", Background_Item
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&Help\tCtrl+?",               Help_Item
        MENUITEM SEPARATOR
        MENUITEM "&About WiniEdit...",          About_Item
    END
END

//////////////////////////////////////////////////////////////////////////////
//
// Icon
//

ProgIcon_ID             ICON    DISCARDABLE     "WiniEdit.ico"

//////////////////////////////////////////////////////////////////////////////
//
// Accelerator
//

Accel_ID ACCELERATORS PRELOAD DISCARDABLE
BEGIN
    "A",            SelectAll_Item,         VIRTKEY, CONTROL, NOINVERT
    "B",            Background_Item,        VIRTKEY, CONTROL, NOINVERT
    "C",            Copy_Item,              VIRTKEY, CONTROL, NOINVERT
    "D",            Default_Item,           VIRTKEY, CONTROL, NOINVERT
    "F",            Format_Item,            VIRTKEY, CONTROL, NOINVERT
    "N",            New_Item,               VIRTKEY, CONTROL, NOINVERT
    "O",            Open_Item,              VIRTKEY, CONTROL, NOINVERT
    "P",            Print_Item,             VIRTKEY, CONTROL, NOINVERT
    "P",            Setup_Item,             VIRTKEY, CONTROL, ALT, NOINVERT
    "Q",            Exit_Item,              VIRTKEY, CONTROL, NOINVERT
    "R",            Revert_Item,            VIRTKEY, CONTROL, NOINVERT
    "S",            Save_Item,              VIRTKEY, CONTROL, NOINVERT
    "S",            SaveAs_Item,            VIRTKEY, CONTROL, ALT, NOINVERT
    "V",            Paste_Item,             VIRTKEY, CONTROL, NOINVERT
    VK_BACK,        Undo_Item,              VIRTKEY, ALT, NOINVERT
    VK_DELETE,      Cut_Item,               VIRTKEY, SHIFT, NOINVERT
    VK_F1,          Help_Item,              VIRTKEY, NOINVERT
    VK_F2,          Cut_Item,               VIRTKEY, NOINVERT
    VK_F3,          Copy_Item,              VIRTKEY, NOINVERT
    VK_F4,          Paste_Item,             VIRTKEY, NOINVERT
    VK_HELP,        Help_Item,              VIRTKEY, CONTROL, NOINVERT
```

Listing 2–1. WiniEdit resource description file (*continued*)

```
    VK_HELP,        Help_Item,              VIRTKEY, SHIFT, CONTROL,
                                                    NOINVERT
    VK_INSERT,      Copy_Item,              VIRTKEY, CONTROL, NOINVERT
    VK_INSERT,      Paste_Item,             VIRTKEY, SHIFT, NOINVERT
    "W",            Close_Item,             VIRTKEY, CONTROL, NOINVERT
    "X",            Cut_Item,               VIRTKEY, CONTROL, NOINVERT
    "Z",            Undo_Item,              VIRTKEY, CONTROL, NOINVERT
END

/////////////////////////////////////////////////////////////////////////////
//
// String Table
//

STRINGTABLE DISCARDABLE
BEGIN
    ProgName_Str            "WiniEdit"
    NoTitle_Str             "Untitled"
    FileFilter_Str          "WiniEdit files (*.wed)|*.wed|Plain text files (*.txt)|*.txt|ASCII files
(*.asc)|*.asc*|All text files (*.wed, *.txt, *.asc)|*.wed;*.txt;*.asc|All files (*.*)|*.*||"
    FileExt_Str             "wed"
END

STRINGTABLE DISCARDABLE
BEGIN
    Save_Msg                "Save document ""%s"" before closing?"
    Revert_Msg              "Revert to most recently saved version of document ""%s""?"
    DefaultFormat_Msg       "Revert document ""%s"" to standard text format?"
    WrongType_Msg           "Sorry, WiniEdit works with text documents only. Can't read or write document
""%s""."
    TooLong_Msg             "Sorry, document ""%s"" is too long for WiniEdit to read."
    OutOfMem_Msg            "Out of memory!"
    IOError_Msg             "Unanticipated input/output error #%s."
END

#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//

/////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

As you compose your resources with the onscreen editors, you can assign them any names and ID numbers you choose. In addition to the main resource description file, the editors also generate a resource header file containing C definitions for the resource names and IDs you've chosen, which you can incorporate into your program's source code with a C **`#include`** directive. Listing 2–2 shows WiniEdit's resource header file. In this case (unlike the resource description file in Listing 2–1), I've done some editing on the raw output produced by the development software, just to pretty the file up a bit for the benefit of my human readers—things like grouping the resource IDs by type instead of in strict numerical order the way the development software produces them. But again, the essential contents of the file were generated automatically by the development system, and again, they include a few arcane incantations that I don't really understand and don't care to—as long as they work (which they do). After including the contents of this file in my source code with the directive

```
#include "WiniEdit Resources.h"
```

I can use the resource IDs it defines to load my resources with statements like

```
resourceID = MAKEINTRESOURCE(ProgIcon_ID);              // Convert resource ID
progIcon   = LoadIcon(ThisInstance, resourceID);        // Load icon
```

and

```
resourceID = MAKEINTRESOURCE(Accel_ID);                 // Convert resource ID
AccelTable = LoadAccelerators(ThisInstance, resourceID); // Load accelerator table
```

and

```
LoadString (ThisInstance, NoTitle_Str,                  // Get default title
            NoNameTitle, TitleMax);
```

Notice that each of these examples uses the variable **`ThisInstance`** to identify the resource module in which to look for the desired resource. This is a global program variable that WiniEdit uses to hold a handle to the currently running instance of the program. Supplying this handle as the resource module parameter tells the Windows resource functions to look for the requested resource in the program's executable file, **`WiniEdit.exe`**.

Listing 2–2. WiniEdit resource header file

```
//
//                              WiniEdit Resources.h
//             Resource header for example Windows application program
//                      S. Chernicoff         15 January 1995
//

//       Global resource header file for WiniEdit example application program

//---------------------------------------------------------------------------

// Icon

#define ProgIcon_ID                     1000

//---------------------------------------------------------------------------

// Menus

#define Main_Menu                       1000

#define File_Menu                   0
#define    New_Item                     1001
#define    Open_Item                    1002
#define    Close_Item                   1003
#define    Save_Item                    1004
#define    SaveAs_Item                  1005
#define    Revert_Item                  1006
#define    Setup_Item                   1007
#define    Print_Item                   1008
#define    Exit_Item                    1009

#define Edit_Menu                   1
#define    Undo_Item                    1101
#define    Cut_Item                     1102
#define    Copy_Item                    1103
#define    Paste_Item                   1104
#define    Delete_Item                  1105
#define    SelectAll_Item               1106

#define Format_Menu                 2
#define    Format_Item                  1201
#define    Default_Item                 1202
#define    Background_Item              1203

#define Help_Menu                   3
#define    Help_Item                    1301
#define    About_Item                   1302

//---------------------------------------------------------------------------
```

Listing 2–2. WiniEdit resource header file (*continued*)

```
// Accelerators

#define Accel_ID                        1000

//----------------------------------------------------------------------------

// Control ID

#define Edit_Control                    1000

//----------------------------------------------------------------------------

// Dialog

#define About_Dialog                    1000

//----------------------------------------------------------------------------

// Strings

#define ProgName_Str                    1001
#define NoTitle_Str                     1002
#define FileFilter_Str                  1003
#define FileExt_Str                     1004

#define Save_Msg                        2001
#define Revert_Msg                      2002
#define DefaultFormat_Msg               2003
#define WrongType_Msg                   2004
#define TooLong_Msg                     2005
#define OutOfMem_Msg                    2006
#define IOError_Msg                     2007

//----------------------------------------------------------------------------

// Next default values for new objects

#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        102
#define _APS_NEXT_COMMAND_VALUE         40004
#define _APS_NEXT_CONTROL_VALUE         1000
#define _APS_NEXT_SYMED_VALUE           102
#endif
#endif
```

The Same Thing...

Processes and Threads

- The Macintosh supports multitasking under System 7 (or earlier system versions with MultiFinder).
- Win32 supports multitasking.

Memory

- The Macintosh has a linear, unsegmented address space.
- Win32 has a linear, unsegmented address space.
- A Macintosh program can allocate blocks of memory from the heap as needed.
- A Win32 program can allocate blocks of memory from the heap as needed.
- A Macintosh program can have multiple independent heap zones.
- A Win32 program can have multiple independent heaps.

File Input/Output

- Macintosh programs can read or write files sequentially with the Toolbox routines **`FSRead`** and **`FSWRite`** (or **`PBRead`** and **`PBWrite`**).
- Win32 programs can read or write files sequentially with the Windows functions **`ReadFile`** and **`WriteFile`**.

Resources

- Macintosh programs use resources to allow aspects of the program's operation to be modified without changing its executable code.
- Win32 programs use resources to allow aspects of the program's operation to be modified without changing its executable code.
- A Macintosh resource is identified by a resource type and either a name or an integer ID number.
- A Win32 resource is identified by a resource type and either a name or an integer ID number.
- On the Macintosh, commonly used system resources are globally available in the system resource file.
- In Win32, commonly used system resources are globally available in the system resource modules.

- On the Macintosh, you can compile resources from a text description with the resource compiler Rez, or create them interactively on-screen with the resource editor ResEdit.
- In Win32, you can compile resources from a text description with the Visual C++ resource compiler, or create them interactively on-screen with specialized resource editors available in the Visual C++ software development environment.

...Only Different

Processes and Threads

- The Macintosh uses cooperative multitasking.
- Win32 uses preemptive multitasking.
- No more than one instance of a Macintosh program can be active at a time.
- Win32 can run multiple instances of a program at the same time, each in its own process.
- To allow the user to view multiple documents at the same time, a Macintosh program must maintain multiple windows on the screen, one for each document.
- A Windows program needn't be able to open more than one document window at a time; the user can view multiple documents by running multiple instances of the program.
- A Macintosh program has a single thread of execution.
- A Win32 program can spawn multiple threads of execution.

Memory

- On the Macintosh, virtual memory is an option available only on some hardware models.
- In Win32, virtual memory is an integral part of the system.
- On the Macintosh, the user chooses whether to enable virtual memory, using the Memory control panel.
- In Win32, virtual memory is always enabled and cannot be turned off.
- On the Macintosh, all virtual memory allocation is handled transparently by the system.
- In Win32, a running program can explicitly reserve and release regions of its address space and commit and decommit physical pages to them.
- The Macintosh Memory Manager can compact the heap by relocating or purging blocks to create more usable free space.
- The Win32 Memory Manager cannot compact the heap; it can only expand it by committing additional pages of physical storage to it.

• Blocks in the Macintosh heap can be either relocatable or nonrelocatable, purgeable or nonpurgeable.	• Blocks in the Win32 heap are always immovable and nondiscardable.
• A block in the Macintosh heap may be identified by either a handle or a simple pointer, depending on whether it is relocatable or nonrelocatable.	• A block in the Win32 heap is always identified by a simple pointer.
• A Macintosh handle is an indirect reference to the underlying data structure: a pointer to the object's master pointer.	• A Win32 handle is an opaque reference to the underlying data structure: its internal contents are meaningful only to the Windows system itself.
• By dereferencing a Macintosh handle twice, you can gain access to the underlying data structure and manipulate its internal fields directly.	• The only way to manipulate the internal fields of a Win32 data structure is indirectly, via a Windows function provided for the purpose.
• On the Macintosh, alternate heap zones are created from within the existing application heap.	• In Win32, alternate heaps occupy separate regions of the process's address space.
• Blocks in the Macintosh heap are always allocated implicitly from the current heap zone. To use a different heap zone, you have to make the desired zone current.	• Win32 heap allocation functions take a handle to a heap as an explicit parameter.
• On the Macintosh, every page of virtual memory is backed by the system's paging file on the hard disk.	• In Win32, a page of virtual memory may be backed by any file on the hard disk.

File Input/Output

• Macintosh programs can only read or write a file sequentially, using **`FSRead`** and **`FSWrite`** (or **`PBRead`** and **`PBWrite`**).	• Win32 programs can read or write a file either sequentially, using **`ReadFile`** and **`WriteFile`**, or randomly, by mapping the file directly into a region of the program's virtual address space.

Program Loading and Execution

- On the Macintosh, if you want to swap parts of your program's code in and out of memory during execution, you must explicitly break up the program into separate code segments, which reside on the disk as separate `'CODE'` resources.
- A Win32 program has no explicit code segments, just a linear stream of code in the program's executable (`.exe`) file.
- The Macintosh system must physically load each of a program's code segments into memory from the disk in order to execute it.
- The Windows system simply maps the code from a program's executable file into its virtual address space, allowing the virtual memory system to load individual pages as needed.
- The executable code of the Macintosh Toolbox resides permanently in read-only memory.
- The executable code of the Windows system resides on the disk in a set of dynamic link libraries.
- A Macintosh program calls a Toolbox routine by using the processor's emulator trap mechanism.
- A Windows program calls a Windows function by mapping the appropriate dynamic link library into its virtual address space and then jumping to the function as an ordinary subroutine.

Resources

- The Macintosh Toolbox has hundreds of built-in resource types.
- Win32 has only about a dozen built-in resource types.
- A Macintosh file consists of two separate parts, a resource fork and a data fork.
- A Win32 file has no resource fork; resources reside in the main body of the file.
- The Macintosh Toolbox searches for a requested resource in a chain of open resource files, beginning with the current resource file.
- Win32 searches for a requested resource only in a single designated file, called a resource module. There is no concept of a current resource file.
- A Macintosh resource type is identified by a four-character name.
- A Win32 resource type is identified by either a character-string name or an integer ID number.
- On the Macintosh, finding a resource in its resource file and loading it into memory are combined into a single operation.
- In Win32, finding a resource in its resource module and loading it into memory are two separate operations, though often they are both performed by a single Windows function.

- On the Macintosh, after creating a resource interactively with the resource editor ResEdit, you can convert it into an equivalent text description with the resource decompiler DeRez.
- In Win32, the Visual C++ interactive resource editors automatically generate a resource description in text form, ready to be compiled with the resource compiler.

Chapter 3

The Windows Programming Model

The fundamental unit of Windows programming is—you guessed it—the window. Windows are even more central to Windows programming than they are to the Macintosh. Everything that happens in a Windows program—not only interactions with the user, but all communications with the system as well—takes place under the control of a window. Just about everything the Windows user sees on the screen is actually a window. Even things that Macintosh programmers aren't accustomed to thinking of as windows, such as icons and menus and pushbuttons and scroll bars, turn out to be just different kinds of window. If you want to understand Windows, you have to begin by understanding windows.

A window communicates with the world by receiving and sending *messages.* Windows messages are much like Macintosh events, in that they report occurrences or circumstances that the window may wish to respond to. In fact, some Windows messages correspond more or less exactly to equivalent Macintosh events: mouse clicks, keystrokes, window activations and updates, and the like. However, as we'll see, Windows also uses the message mechanism to signal higher-level conditions that a Macintosh program could obtain only through additional calls to the Toolbox. The idea of sending a message to a window is the central unifying concept on which the entire Windows programming model is based.

Each window has a *window procedure* to handle the messages it receives from the system. Every message sent to a window ultimately finds its way to the window procedure for processing. By defining how a window responds to messages, the window procedure determines everything about the window's appearance and behavior on the screen. A category of windows sharing the same window procedure is known as a *window class.* Every window must belong to some class or other, specified as a parameter when the window is created. A typical program defines at least one class, for its main window, and may define additional classes for other kinds of window that it uses on the screen.

One of the neat things about Windows programming is that there's already a complete window procedure built into the system and ready to use. This predefined window procedure, named **`DefWindowProc`** ("default window procedure"), provides a standard response for each type of message your windows can receive. This saves you from having to write your own code for every one of the hundreds of possible messages the system can throw at you: you can just "cherry-pick" those messages you care about and pass the rest on to the default window procedure, knowing that it will do something sensible with them.

Using the default window procedure gives Windows programming something of an object-oriented flavor, even if you're not actually using an object-oriented application framework like the Microsoft Foundation Classes. In effect, the default window procedure defines a universal

superclass that determines the baseline appearance and behavior for all windows. Writing your own window procedure is like creating a subclass: your windows inherit all of the standard behavior except for those messages that you explicitly override with code of your own.

Message Structure

Each message is represented by a *message structure* of type **MSG**, shown in Listing 3–1. If the contents of this structure look familiar to you, it's because they're essentially the same as the fields of the Macintosh event record: the type of message, the time it was posted, the coordinates of the mouse at the time, and some additional parameter information that varies depending on the message type. Notice, though, that the Windows message also has to identify the window to which it is directed; this isn't necessary on the Macintosh, where all events are addressed directly to the program itself. Notice also that the Windows message contains *two* type-dependent parameters instead of only one. Originally, one of these was a 16-bit word (**wParam**) and the other a 32-bit long word (**lParam**). In the Win32 interface, however, the data types **WPARAM** and **LPARAM** are both defined as equivalent to **LONG**, so both parameters are actually 32 bits in length.

When it comes to sheer variety of messages, Windows runs rings around the Macintosh Toolbox. The Macintosh makes do with only sixteen event types, mainly representing low-level user actions (mouse-down, mouse-up, key-down, key-up, auto-key, disk-inserted) and very basic window manipulations (activate, update, suspend, resume). Even the new-fangled Apple events (which pertain only to communicating with other programs anyway) are subsumed under one single event type, "high-level events." Windows, on the other hand, defines literally hundreds of message types—393 at last count, with more being added all the time. Luckily for Windows programmers, it isn't necessary for every program to provide an explicit response to all 393 messages. As we'll see, Windows has a built-in window procedure that already provides a reasonable way of dealing with most of them; the program needn't step in unless it wants to handle something in an unusual or nonstandard way.

Each message type is identified by its own unique *message identifier* in the **message** member of the **MSG** structure. The Win32 header files define constant names for each of the possible message identifiers, such as **WM_CREATE**, **WM_MOVE**, **WM_KEYDOWN**, and **WM_MENUSELECT**. (Many of the most common message identifiers begin with the prefix **WM_**, for "window message"; there are also other categories, such as **BM_** for "button message," **LB_** for "list box," and so on, many of which we'll be covering in later chapters.) Table 3–1 shows a few of the more common

Listing 3–1. Message structure

```
typedef struct tagMSG
   {
      HWND    hwnd;                    // Handle to window receiving message
      UINT    message;                 // Message identifier
      WPARAM  wParam;                  // Type-dependent information
      LPARAM  lParam;                  // Type-dependent information
      DWORD   time;                    // Time message was posted
      POINT   pt;                      // Mouse position at time of message

   }  MSG;
```

Table 3–1. Some typical message types

Message type	Meaning
WM_CREATE	Window being created; initialize its associated data structures
WM_DESTROY	Window being destroyed; deallocate or finalize its data structures
WM_SHOWWINDOW	Show or hide window on screen
WM_CLOSE	Close window
WM_ACTIVATE	Window being activated or deactivated
WM_SIZE	Window's size has changed
WM_LBUTTONDOWN	Left mouse button pressed
WM_CHAR	Character typed on keyboard
WM_COMMAND	Command chosen from menu
WM_QUIT	Terminate program

message types and their meanings.

Queued and Unqueued Messages

Windows maintains a *message queue* for messages pending delivery, similar to the event queue on the Macintosh. Each running program (and in a multithreaded program, each separate thread) has its own message queue, holding messages addressed to that program's windows. The low-level device drivers for the mouse and keyboard gather the user's input actions and post them as messages to a single, system-wide queue; the Windows system then pulls them out one at a time, figures out which window they're addressed to, and reposts them into the private message queue of the program or thread that created that window. Eventually, the program's messasge loop wll retrieve each message and dispatch it to the appropriate window procedure for processing. In general, the queue is reserved for messages reporting "raw" user input like mouse clicks and keystrokes; it also receives a few other types of message such as **WM_TIMER** (a software-controlled interval timer has expired), **WM_PAINT** (the window must redraw all or part of its contents, similar to a Macintosh update event), and **WM_QUIT** (the program is about to terminate). Table 3–2 summarizes the message types that are posted to the queue. We'll be talking about a few of them in this chapter; the ones relating to mouse and keyboard input will be covered in detail in Chapter 5.

Table 3–2. Queued messages

Message type	Meaning
WM_LBUTTONDOWN	Left mouse button pressed in client area (like Macintosh content region)
WM_LBUTTONUP	Left mouse button released in client area
WM_LBUTTONDBLCLK	Left mouse button double-clicked in client area
WM_MBUTTONDOWN	Middle mouse button pressed in client area
WM_MBUTTONUP	Middle mouse button released in client area
WM_MBUTTONDBLCLK	Middle mouse button double-clicked in client area
WM_RBUTTONDOWN	Right mouse button pressed in client area
WM_RBUTTONUP	Right mouse button released in client area
WM_RBUTTONDBLCLK	Right mouse button double-clicked in client area
WM_NCLBUTTONDOWN	Left mouse button pressed in nonclient area (like Macintosh window frame)
WM_NCLBUTTONUP	Left mouse button released in nonclient area
WM_NCLBUTTONDBLCLK	Left mouse button double-clicked in nonclient area
WM_NCMBUTTONDOWN	Middle mouse button pressed in nonclient area
WM_NCMBUTTONUP	Middle mouse button released in nonclient area
WM_NCMBUTTONDBLCLK	Middle mouse button double-clicked in nonclient area
WM_NCRBUTTONDOWN	Right mouse button pressed in nonclient area
WM_NCRBUTTONUP	Right mouse button released in nonclient area
WM_NCRBUTTONDBLCLK	Right mouse button double-clicked in nonclient area
WM_MOUSEMOVE	Mouse position changed within client area
WM_NCMOUSEMOVE	Mouse position changed within nonclient area
WM_KEYDOWN	Key pressed on keyboard (no Alt key)
WM_KEYUP	Key released on keyboard (no Alt key)
WM_SYSKEYDOWN	Key pressed on keyboard in combination with Alt key
WM_SYSKEYUP	Key released on keyboard in combination with Alt key
WM_CHAR	Character typed on keyboard (no Alt key)
WM_DEADCHAR	Dead character typed on keyboard (no Alt key)
WM_SYSCHAR	Character typed on keyboard in combination with Alt key
WM_SYSDEADCHAR	Dead character typed on keyboard in combination with Alt key
WM_TIMER	Software-controlled interval timer expired
WM_PAINT	Redraw all or part of client area
WM_QUIT	Terminate program

Unlike Macintosh events, however, not all Windows messages actually go through the queue. Many of the messages a window receives are *unqueued* messages: the system sends them directly to the relevant window procedure instead of posting them to the queue to be retrieved by the program's message loop. Whereas queued messages generally correspond to the same kinds of low-level user input as Macintosh events, unqueued messages operate at a higher level of discourse, reporting conditions and occurrences that have been generated or interpreted within the Windows system itself. Often, the system's response to one message will trigger a whole series of further messages sent directly to the window procedure. These high-level, unqueued messages are what gives Windows programming much of its ease and power.

Consider what happens when a Macintosh program receives a mouse-down event from the Toolbox Event Manager. The decision logic needed to interpret and respond to such an event typically involves an elaborate sequence of Toolbox calls designed to narrow down the location and meaning of the event and provide the appropriate user feedback on the screen. When the user presses and drags the mouse in a window's size box, for example, it touches off a sequence of operations more or less like the following:

1. Your main event loop receives the event from **`WaitNextEvent`**.
2. You examine the event code in the **`what`** field of the event record and determine that it is a mouse-down event.
3. You pass the event to the Toolbox routine **`FindWindow`** to determine where on the screen the mouse was pressed, and learn that it was in the size box of the active window.
4. You call the Toolbox routine **`GrowWindow`** to track the mouse for as long as the user continues to hold down the button, providing visual feedback in the form of a dotted outline whose bottom-right corner follows the mouse's movements.
5. When the button is released, **`GrowWindow`** returns to you with the new dimensions of the window. You then pass this information to the Toolbox routine **`SizeWindow`** to resize the window to its new dimensions.
6. **`SizeWindow`** automatically redraws the window's frame in its new size, as well as any portions of other windows' frames that have become visible as a result of the resizing operation. It also generates update events for those portions of any window's content region that need to be redrawn.
7. On a later pass of your event loop, you will receive the update event(s) and will redraw the contents of the window(s) as needed.

Now let's look at how Windows would handle this same event. Although the Windows mouse can have as many as three buttons, the usual convention is to use the left button for manipulating windows on the screen; so the Windows equivalent to the user's pressing the mouse button in a Macintosh window's size box would be to press the left button in the window's sizing border. The sizing border is part of the window's nonclient area—the part of the window that the system draws for you, corresponding to the window frame in Macintosh terminology—so this will generate a **`WM_NCLBUTTONDOWN`** message (**`NC`** for "nonclient," **`L`** for "left") analogous to the Macintosh mouse-down event. Your message loop will retrieve this message from the queue with the Windows function **`GetMessage`** (like **`WaitNextEvent`** on the Macintosh) and hand it off to another Windows function, **`DispatchMessage`**; the latter, in turn, will relay it back to your program's window procedure.

At this point, your window procedure could (theoretically, anyway) choose to process the **WM_NCLBUTTONDOWN** message "by hand," with a sequence of operations similar to the ones you'd need to perform on the Macintosh: hit-testing the mouse location, tracking the mouse as it drags the window's sizing border, resizing the window, and so on. But it's much easier to skip all that and just pass the message on to the default window procedure, **DefWindowProc**. That's what the default window procedure is there for: to provide a standard response to each message you receive, saving you the trouble of having to write one for yourself.

So let's assume you pass that mouse message (**WM_NCLBUTTONDOWN**) to the default window procedure. Here's where things start to get interesting. The first thing **DefWindowProc** has to do is figure out what part of the window's nonclient area received the mouse click, so it sends the window the message **WM_NCHITTEST**, with the mouse's screen coordinates as a parameter. Again, you could intercept this message in your window procedure and process it yourself, but you'll usually just pass it through to the default window procedure to perform the hit-test in the standard way.

When the result of the **WM_NCHITTEST** message indicates that the mouse press was in the window's sizing border, the default procedure will track the mouse by processing **WM_NCMOUSEMOVE** messages until it receives a **WM_NCLBUTTONUP**. Each time it receives a **WM_NCMOUSEMOVE**, it calls the Windows function **MoveWindow** to provide visual feedback to the user by redrawing the window in a new size. **MoveWindow** redraws the window's nonclient area (frame) and then sends another series of messages back to the window:

- **WM_MOVE** to tell it the new coordinates of its top-left corner
- **WM_SIZE** to tell it its new width and height
- **WM_NCCALCSIZE** to ask it to recalculate the coordinates of its client area (the Windows equivalent of the Macintosh content region)
- **WM_PAINT** to tell it to redraw the contents of its client area

There's also a pair of "hook" messages to allow the window procedure to get control before and after redrawing, in case it wants to customize the operation in some way:

- **WM_WINDOWPOSCHANGING** to tell it its size or position is about to change
- **WM_WINDOWPOSCHANGED** to tell it its size or position *has* just changed

(If the window procedure intercepts the **WM_WINDOWPOSCHANGED** message, the later **WM_MOVE** and **WM_SIZE** messages are suppressed, since by that time the window has already had a chance to adjust to its new position and size.)

If all this proliferation of messages strikes you as rather complex, you're right. Keep in mind, though, that you're not required to deal with all of them yourself, because the default window procedure is always there to back you up. In fact, most of these messages are really *intended* to be handled by the default window procedure, beneath your program's level of notice. The only reason for sending the messages is to give you the *option*, at various points in the process, of stepping in and taking control for yourself. The combination of the message mechanism with the default window procedure gives you the freedom to sit back and let the Windows system handle everything for you, together with the flexibility to redefine or customize its behavior if you choose to do so.

A Simple Windows Program

One of the common criticisms of the Macintosh Toolbox, right from the beginning, has been the amount of coding effort needed to get even the simplest application running. The Toolbox provides all the support needed to put windows on the screen, move them, size them, scroll them, and so on; but there's still a considerable burden on the program (and the programmer) to issue the right sequence of Toolbox calls to gather the user's mouse clicks, interpret them, and carry them out. By contrast, Windows' high-level message structure and built-in default window procedure make it comparatively easy to get a simple application running. Listing 3–2, for example, shows the complete code for a "null" application that just puts a window on the screen and lets the user manipulate it with the mouse. The window has all the expected paraphernalia that we'll be discussing in Chapter 4—a title bar for dragging it around the screen, a sizing border, maximize and minimize boxes, a system menu—and they all work just the way the user expects them to. If you throw out all the comments, white space, and pretty-printing (not to mention my somewhat verbose coding style), the entire program comes to only about 50 lines of "live" code. An equivalent Macintosh program (including support for desk accessories, which don't exist in Windows) would have to be at least five times that long.

Main Function

The main entry point to our NullApp program is named **`WinMain`**. Every Windows program must begin with a function of this name, which takes four parameters:

- A handle to the current instance of the program, the one just being started up. (Recall from Chapter 2 that Windows allows more than one instance of a program to run at the same time.)
- If any other instances of the same program are already running, a handle to the previous instance most recently started before this one. If the current instance is the only one now running, this parameter is **`NULL`**.
- A pointer to a null-terminated character string containing any parameters being passed to the program via its command line. This parameter is a relic of the nasty old DOS command-line interface; in Windows, programs are normally started with a double click or a menu command from the Program Manager or File Manager, rather than by typing a command line. If the user starts the program by double-clicking one of its document files, the system will pass the document's path name in this parameter.

Listing 3–2. A null Windows application

```
//
//                                  NullApp
//                          Null Windows application program
//                        S. Chernicoff         15 January 1995
//

//         Simple program to display and manipulate a window on the screen


   #include <windows.h>


//-------------------------------------------------------------------------------------------------------------


// Global variables


   HWND  TheWindow;                                    // Handle to main window
   BOOL  ContinueFlag = TRUE;                          // Keep running?

//-------------------------------------------------------------------------------------------------------------


// Function prototypes


INT CALLBACK WinMain (HINSTANCE thisInstance, HINSTANCE prevInstance, CHAR *commandLine, INT showState);
      // Main function
   VOID PASCAL Initialize (HANDLE thisInstance, HANDLE prevInstance, CHAR *commandLine, INT showState);
         // Do one-time-only initialization.
      VOID InitClass (HANDLE thisInstance, HANDLE prevInstance);
            // Register the window class.
      VOID InitWindow (HANDLE thisInstance, INT showState);
            // Create the window.
   VOID MainLoop (VOID);
         // Execute one pass of main program loop.
   VOID Finalize (VOID);
         // Do one-time-only finalization.

//-------------------------------------------------------------------------------------------------------------
```

Listing 3–2. A null Windows application (*continued*)

```
INT CALLBACK WinMain (HINSTANCE thisInstance, HINSTANCE prevInstance, CHAR *commandLine, INT showState)

   // Main function

   {

      Initialize (thisInstance, prevInstance, commandLine, showState);  // Do one-time-only initialization


      do
         MainLoop ();                              // Execute one pass of main loop
      while ( ContinueFlag );                      // Continue until time to quit


      Finalize ();                                 // Do one-time-only finalization


      return NO_ERROR;                             // Signal successful completion


   }  /* end WinMain */

//-------------------------------------------------------------------------------------------------------

VOID PASCAL Initialize (HANDLE thisInstance, HANDLE prevInstance, CHAR *commandLine, INT showState)

   // Do one-time-only initialization.

   {
      InitClass  (thisInstance, prevInstance);  // Register the window class
      InitWindow (thisInstance, showState);     // Create the window

   }  /* end Initialize */

//-------------------------------------------------------------------------------------------------------
```

Listing 3–2. A null Windows application (*continued*)

```
VOID InitClass (HANDLE thisInstance, HANDLE prevInstance)

   // Register the window class.

   {
      WNDCLASS  windowClass;                                  // Window class
      HICON     progIcon;                                     // Program's screen icon
      HCURSOR   arrowCursor;                                  // Default cursor
      HBRUSH    bkBrush;                                      // Brush for painting window background


      if ( prevInstance == NULL )                             // Is this the first instance of program?
         {
            windowClass.lpszClassName = "NullApp";            // Use program name as class name

            windowClass.lpfnWndProc = DefWindowProc;          // Use default window procedure
            windowClass.hInstance   = thisInstance;           // Current instance is the owner

            windowClass.style = NULL;                         // No special style

            windowClass.lpszMenuName = NULL;                  // No menu

            progIcon = LoadIcon(NULL, IDI_APPLICATION);       // Load generic icon
            windowClass.hIcon = progIcon;                     // Set icon

            arrowCursor = LoadCursor(NULL, IDC_ARROW);        // Load stock arrow cursor
            windowClass.hCursor = arrowCursor;                // Set cursor

            bkBrush = HBRUSH(COLOR_WINDOW + 1);               // Create brush for system background color
            windowClass.hbrBackground = bkBrush;              // Set window background brush

            windowClass.cbClsExtra = 0;                       // No extra class data
            windowClass.cbWndExtra = 0;                       // No extra window data

            RegisterClass (&windowClass);                     // Register the class

         }  /* end if ( PrevInstance == NULL ) */

   }  /* end InitClass */

//--------------------------------------------------------------------------------------------------
```

Listing 3–2. A null Windows application (*continued*)

```
VOID InitWindow (HANDLE thisInstance, INT showState)

   // Create the window.

   {
      TheWindow = CreateWindow ("NullApp",               // Use program name for class name
                                "This space left blank", // Set window title
                                WS_OVERLAPPEDWINDOW,     // Use standard window style
                                CW_USEDEFAULT,           // Let Windows choose initial x
                                CW_USEDEFAULT,           //   and y position
                                CW_USEDEFAULT,           // Let Windows choose initial width
                                CW_USEDEFAULT,           //   and height
                                NULL,                    // No parent window
                                NULL,                    // Use menu from window class
                                thisInstance,            // Use current program instance
                                NULL);                   // No special creation parameters

      ShowWindow   (TheWindow, showState);               // Display window on screen
      UpdateWindow (TheWindow);                          // Force update of client area

   }  /* end InitWindow */

//-----------------------------------------------------------------------------------------------------

VOID MainLoop (VOID)

   // Execute one pass of main program loop.

   {
      MSG   theMessage;                             // Next message to process


      ContinueFlag = GetMessage(&theMessage, NULL, 0, 0);   // Get next message

      TranslateMessage (&theMessage);               // Convert virtual keys to characters
      DispatchMessage  (&theMessage);               // Send message to window procedure

   }  /* end MainLoop */

//-----------------------------------------------------------------------------------------------------

VOID Finalize (VOID)

   // Do one-time-only finalization.

   {
      DestroyWindow (TheWindow);                    // Destroy the window

   }  /* end Finalize */
```

- An integer code specifying how the program is to display its main window at startup. The value of this parameter is normally one of two constants defined in the Windows API header files: either **SW_SHOWNORMAL**, meaning to display the window in the normal way, or **SW_SHOWMINNOACTIVE**, meaning to show the window initially in its minimized state (as a button in the Windows 95 task bar, or as an icon on the desktop in earlier versions of Windows). After creating its main window, the program typically just passes the value of this parameter along to the Windows function **ShowWindow**, which will display the window in the requested state.

Notice in Listing 3–2 that the **WinMain** function is declared with the keyword **CALLBACK**. This keyword is defined in the Windows API header files and is required for any function (such as a main entry point or a window procedure) that will be called directly by the Windows system, rather than from within the program itself.

Looking at the logical structure of our **WinMain** function, you'll see that it's no different from that of a typical Macintosh program. In fact, if you compare it with the main function of my old MiniEdit program in Appendix A, you'll find that it's virtually identical. Just as you would expect, the program begins by doing some one-time initialization, then enters a main loop that it executes repeatedly under the control of a boolean flag variable. In a Macintosh program, each pass of the main loop processes one event; in a Windows program, each pass processes one message. When something happens to change the value of the control flag, the program falls out of its main loop, does any needed finalization, and exits back to the system shell.

One minor difference you may notice between the Macintosh and Windows versions of the program is the polarity of the control flag. The Macintosh version uses a global boolean flag named **Finished**, which is initialized to **false** and becomes **true** when the user chooses the **Quit** command from the menu; this causes the program to fall out of its main loop and terminate. In the Windows version, the name of this flag is changed to **ContinueFlag** and its polarity is reversed: it is initially **TRUE** and becomes **FALSE** to signal termination. This change of polarity is just a minor programming convenience resulting from the way the Windows system signals when it's time for the program to quit, as we'll see later in this chapter.

Registering a Window Class

One of the first orders of business for any Windows program is to *register* its window classes with the system so that it can use them to create its windows. The Windows function **RegisterClass** accepts a pointer to a data structure of type **WNDCLASS**, which the program fills in with descriptive information about the class to be registered. Listing 3–3 shows the contents of this structure, which include the following items:

- The name of the class.
- A pointer to its window procedure.
- The program instance to which it belongs.
- A word of bit flags specifying various aspects of the way its windows appear or behave.
- An icon for representing its windows on the screen when in their minimized state.
- A default cursor shape to use when the cursor is moved into its windows.
- A brush (like a QuickDraw pattern) for painting the background area inside its windows.

Listing 3–3. Contents of **WNDCLASS** structure

```
typedef struct _WNDCLASS
   {
      UINT     style;              // Style options
      WNDPROC  lpfnWndProc;        // Pointer to window procedure
      int      cbClsExtra;         // Extra bytes of client data in class structure
      int      cbWndExtra;         // Extra bytes of client data in window structure
      HANDLE   hInstance;          // Program instance registering the class
      HICON    hIcon;              // Icon for minimizing windows
      HCURSOR  hCursor;            // Default cursor
      HBRUSH   hbrBackground;      // Brush for painting window background
      LPCTSTR  lpszMenuName;       // Resource name of menu
      LPCTSTR  lpszClassName;      // Name of class

   }  WNDCLASS;
```

- The resource name or ID number of its top-level menu, which defines the contents of its windows' menu bars (actually more like a Macintosh **'MBAR'** than a **'MENU'** resource).
- The number of bytes of extra, private data the program wishes to allocate in the window class structure itself and in each window structure created from it. (We'll learn how to access this extra data in Chapter 4.)

> In earlier versions of the Windows system, multiple instances of the same program weren't allowed to reregister the same window class more than once. The normal way to prevent this was to test the previous-instance handle that the program's main function received as a parameter, and register the class only if the previous instance was **NULL** (that is, if no other instance of the program already existed). Although Win32 has removed this restriction, there's no harm in observing it anyway for backward compatibility, as illustrated by our NullApp program's **InitClass** function in Listing 3–2.

Every window class has to have a unique name, different from that of any other class in the system. To ensure uniqueness, the common convention is to use the name of your program as the name of its main window class. You can see an example of this in the **InitClass** function of Listing 3–2, where the name of the class is set with the statement

```
windowClass.lpszClassName = "NullApp";          // Use program name as class name
```

For the class's window procedure, NullApp simply uses the system's built-in window procedure, **DefWindowProc**, without modification:

```
windowClass.lpfnWndProc = DefWindowProc;        // Use default window procedure
```

This means, of course, that the program's windows can't do anything special beyond the standard behavior that's built into the system by default. This allows the user to use all the standard components of the window (the title bar, sizing border, and so forth) without displaying anything in the window or responding to user input in any way. We'll see later how to write our own window procedure to extend or override the standard behavior.

The s**tyle** member of the **WNDCLASS** structure typifies a common technique that we'll find repeated again and again throughout the Windows programming interface: a word of bit flags that define or modify the way an element of the interface works. Table 3–3 shows the style options that are available for window classes. Each of the names in the table, such as **CS_NOCLOSE**, is defined in the Win32 header files as a constant bit mask corresponding to a single bit in the **WNDCLASS** structure's 16-bit **style** word. You can combine these masks with the C language's bitwise logical operators (**&**, **|**, **^**, and **~**) to form any combination of style bits you need. For instance, although our NullApp program doesn't request any of these options (it just sets **windowClass.style** to **NULL**), we'll see later that WiniEdit uses the statement

```
windowClass.style = CS_HREDRAW | CS_VREDRAW;
```

to request that its windows be redrawn whenever their size changes in either the horizontal or vertical dimension. Other options allow you to specify whether the window has a close box, whether it receives special messages for double mouse clicks, whether it has its own private device context (the Windows equivalent of a QuickDraw graphics port) or inherits one from its parent, and so on. Some of these options are rather obscure, but they're available in case you need them.

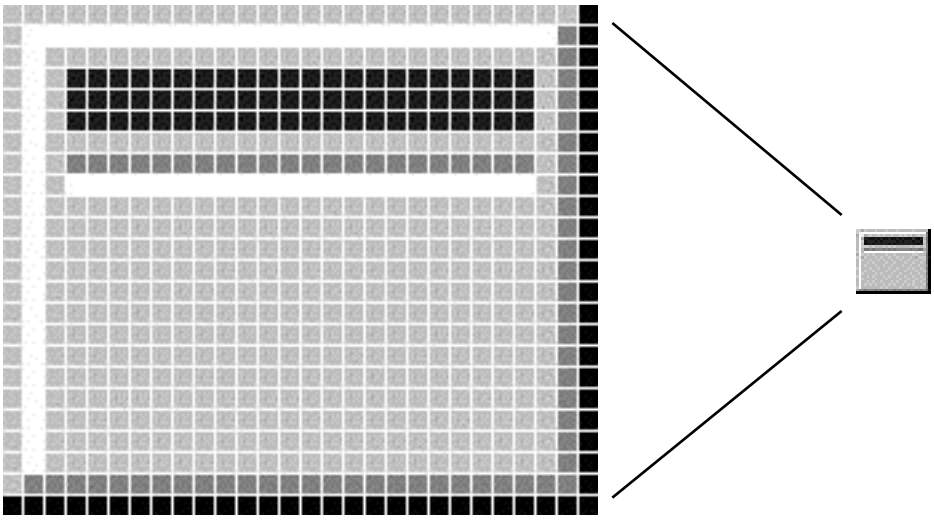
Every window class has three graphical items associated with it: an icon, a cursor, and a background brush. The icon defines how windows belonging to the class appear when minimized. In older versions of Windows, it would be used to represent the window directly on the screen, like a file or folder on the desktop in the Macintosh Finder; in Windows 95, it is displayed in reduced form as part of the button representing the window in the task bar. NullApp just uses the system's standard, "generic" application icon (Figure 3–1). This icon is available as a system resource under the name **IDI_APPLICATION**:

```
progIcon = LoadIcon(NULL, IDI_APPLICATION);        // Load generic icon
windowClass.hIcon = progIcon;                      // Set icon
```

Table 3–3. Window class style options

Style name	Meaning
CS_NOCLOSE	Omit close box and system-menu Close command
CS_DBLCLKS	Receive double-click messages
CS_HREDRAW	Redraw window when width changes
CS_VREDRAW	Redraw window when height changes
CS_BYTEALIGNCLIENT	Align client area on horizontal byte boundaries for faster drawing
CS_BYTEALIGNWINDOW	Align nonclient area on horizontal byte boundaries for faster positioning
CS_SAVEBITS	Save obscured portion of screen for later restoration
CS_OWNDC	Allocate a private device context for each window of class
CS_PARENTDC	Inherit parent window's device context
CS_GLOBALCLASS	Make class globally accessible to all programs

Figure 3–1. Generic application icon



(The prefix **IDI**_stands for "identifier of an icon.") WiniEdit, on the other hand, provides an icon of its own, created and placed in the program's executable file with the Visual C++ onscreen icon editor. The icon's resource ID is defined in the program's resource header file, **WiniEdit Resources.h**, as a constant named **ProgIcon_ID**. The following statements load the resource and associate it with the program's window class:

```
resourceID = MAKEINTRESOURCE(ProgIcon_ID);          // Convert resource ID
progIcon   = LoadIcon(ThisInstance, resourceID);    // Load icon
windowClass.hIcon = progIcon;                       // Set icon
```

The cursor associated with a window class tells the Windows system what cursor shape to use by default when the user moves the mouse into one of the class's windows. (We'll learn in Chapter 5 how a program can use messages from the system to do its own cursor management.) Both NullApp and WiniEdit use the "stock" arrow cursor, retrieved as a system resource:

```
arrowCursor = LoadCursor(NULL, IDC_ARROW);          // Load stock arrow cursor
windowClass.hCursor = arrowCursor;                  // Set cursor
```

A *brush* is the Windows equivalent of a Macintosh pattern: a small pixel image that can be repeated indefinitely to fill an area, like identical floor tiles laid end to end. We'll save our discussion of brushes for Chapter 6, when we talk about the Windows Graphical Device Interface; all we need to know for now is that each window class includes a *background brush* for filling in the background of a window's client area. Just as with icons and cursors, the system defines a set of stock brushes as system resources, in solid black, white, and various shades of gray, as well as a transparent or "hollow" brush.

There's also a set of 21 brushes corresponding to *system colors* that the user is free to set with the Color control panel. These represent the user's preferences for the colors of various user interface elements, such as **COLOR_MENU** for the background color of menus, **COLOR_MENUTEXT** for the text of menu items, **COLOR_SCROLLBAR** for the shafts of scroll bars, and so forth. In particular, the system color **COLOR_WINDOW** is the color the user has chosen for painting the background area of a window. You can get a brush for any of the system colors by typecasting the color's ID

number directly into a brush handle (type **HBRUSH**). However, for reasons unknown to mere mortals, the Windows gods have decreed that the number of the brush is *one more than* the number of the corresponding system color. Thus both NullApp and WiniEdit use the following statements to set the background brush for their window classes:

```
bkBrush = HBRUSH(COLOR_WINDOW + 1);    // Create brush for system background color
windowClass.hbrBackground = bkBrush;   // Set window background brush
```

NullApp's windows don't need any menus, since all they do is let themselves be dragged around or resized on the screen. So the code specifying the menu for the program's window class is simply

```
windowClass.lpszMenuName = NULL;                    // No menu
```

In WiniEdit's case, the contents of the menu bar are defined as a resource in the program's executable file under the resource ID **Main_Menu**. Because the **WNDCLASS** structure expects a string defining the name of the menu resource, the integer resource ID must be converted to the proper form with the **MAKEINTRESOURCE** macro that we discussed in Chapter 2:

```
resourceID = MAKEINTRESOURCE(Main_Menu);            // Convert resource ID
windowClass.lpszMenuName = resourceID;              // Set menu
```

Creating and Displaying a Window

Once you've registered your window class, the next step is to use it to create your main window and display it on the screen. The Windows functions for doing this are **CreateWindow** and **ShowWindow**. We'll just look at them briefly here and save the details for our discussion of windows in Chapter 4.

NullApp uses the following statement to create its window:

```
TheWindow = CreateWindow ("NullApp",            // Use program name for class name
                          "This space left blank",    // Set window title
                          WS_OVERLAPPEDWINDOW,  // Use standard window style
                          CW_USEDEFAULT,        // Let Windows choose initial x
                          CW_USEDEFAULT,        //   and y position
                          CW_USEDEFAULT,        // Let Windows choose initial width
                          CW_USEDEFAULT,        //   and height
                          NULL,                 // No parent window
                          NULL,                 // Use menu from window class
                          thisInstance,         // Use current program instance
                          NULL);                // No special creation parameters
```

Notice that the window's class is identified by name, rather than directly by means of a handle to the class object itself. This is why every window class must have a name that is unique in the entire system. As we've already mentioned, the usual convention is to use the name of the program itself (in this case, **"NullApp"**) as the name of its main window class. The **CreateWindow** function creates a window of the specified class and returns a window handle of type **HWND**.

Recall that one of the parameters your **WinMain** function receives from the system when you first start up is an integer code specifying how to display your main window initially on the screen. This value is known as a *show state*, and is passed as a parameter to the Windows

function **ShowWindow** to tell it how to display the window. Normally, the show state you receive from the system will be the constant **SW_SHOWNORMAL**, meaning to open your window in its normal, fully displayed state. However, if the user has held down the Shift key while starting up the program, or has checked the **Run Minimized** option to the Program Manager's or File Manager's **Run...** command, you'll get a show-state parameter of **SW_SHOWMINNOACTIVE** instead, telling you to display the window in a minimized, inactive state. You don't have to inspect this parameter for yourself, though—just pass it along to **ShowWindow** after creating your main window, as our NullApp program does with the statement

```
ShowWindow (TheWindow, showState);                    // Display window on screen
```

in its **InitWindow** routine, and Windows will do the right thing. After displaying the window, you should also call the Windows function **UpdateWindow**

```
UpdateWindow (TheWindow);                             // Force update of client area
```

to make sure its initial contents get drawn properly. As we'll see in Chapter 4, the result is to generate a **WM_PAINT** message to your window procedure, telling it to draw the contents of the client area—just like a Macintosh update event.

Main Loop

After doing its preliminary initialization, our program's main function, **WinMain**, enters its message loop, in which it repeatedly calls the routine **MainLoop** under the control of a global flag, **ContinueFlag**. As long as this flag remains **TRUE**, the program will continue iterating the loop; when the flag becomes **FALSE**, the loop will terminate and control will fall through into the **Finalize** routine.

Our **MainLoop** routine issues calls to three Windows functions: **GetMessage**, **TranslateMessage**, and **DispatchMessage**. **GetMessage** is the basic message-retrieval function, analogous to **GetNextEvent** or **WaitNextEvent** on the Macintosh. It accepts a pointer to a message structure as its first parameter, fills it in with descriptive information about the message, and removes the message from the queue. The remaining parameters allow you to place restrictions on the kind of message you're requesting. The second parameter is a window handle, limiting the request to messges addressed specifically to that window or one of its children. A **NULL** handle, as in the current example, denotes any window belonging to the current thread (the one issuing the **GetMessage** call). The last two parameters narrow the request to a limited range of (minimum and maximum) message types; they thus serve a purpose similar to (though less flexible than) the event mask parameter to **GetNextEvent** or **WaitNextEvent** on the Macintosh. For example, you can limit your request to mouse messages only by setting these two parameters to the constants **WM_MOUSEFIRST** and **WM_MOUSELAST**, or to keyboard messages by using **WM_KEYFIRST** and **WM_KEYLAST**. Setting both parameters to **0** asks for the next pending message in the queue, regardless of type.

The **GetMessage** function returns a boolean result that's always **TRUE** unless the message being returned is **WM_QUIT**. You can use this boolean value to control the termination of your main message loop, as NullApp does by setting the global **ContinueFlag**. How does Windows know when to send you a **WM_QUIT** message? You tell it when, by calling the Windows function **PostQuitMessage**. (Our WiniEdit program, for instance, calls this function in response to the user's choosing the **Exit** menu command.) The **WM_QUIT** message gives your window

procedure a chance to do any last-minute cleanup it might need before falling out of the message loop.

> You may be wondering what happens if you ask for a message and there aren't any waiting in the queue. Windows has no equivalent to the Macintosh concept of a null event. Instead, the **GetMessage** function simply suspends your thread's execution until a message arrives, allowing other processes or threads to run in your place. When your call to **GetMessage** does eventually return, it's guaranteed to come back with a bona-fide message for you to respond to.
>
> There's also an alternative retrieval function, **PeekMessage**, which always returns immediately, with or without a message to report. Like **GetMessage**, **PeekMessage** returns a boolean result; but whereas **GetMessage** uses its result to signal the arrival of a **WM_QUIT** message, **PeekMessage** uses it to report whether there was a message in the queue to retrieve. If there is a message, **PeekMessage** copies it to the message structure you supply as a parameter and returns a **TRUE** result; if the queue is empty, it returns **FALSE** and leaves the message structure untouched. **PeekMessage** also takes an additional parameter telling it whether to remove any message it returns or leave it in the queue for later processing; the latter option makes it analogous to the Macintosh **EventAvail** function.
>
> Finally, if **PeekMessage** reports that the queue is empty, you can use the Windows function **WaitMessage** to suspend execution of your thread pending the arrival of a message, similarly to what **GetMessage** does when it finds the queue empty.

After receiving a message from **GetMessage**, the next step is to pass it to another Windows function, **TranslateMessage**. This is a rather specialized function that maps raw keystroke messages such as **WM_KEYDOWN** and **WM_SYSKEYDOWN** into corresponding character-based messages like **WM_CHAR** and **WM_SYSCHAR**, under the control of the currently installed keyboard layout; it has no effect on non-keyboard messages. We'll have more to say about **TranslateMessage** when we discuss keyboard input processing in Chapter 5; for now, all we need to know is that every message we receive from **GetMessage** should be passed to **TranslateMessage** before further processing.

The key step after retrieving a message from the queue is to pass it to the Windows function **DispatchMessage**. This examines the message to see which window it's addressed to, looks up the window class to which the window belongs, and relays the message to the window procedure registered for that class. What happens next is up to the window procedure, which is, of course, part of your program that you write yourself. Our NullApp program doesn't have a window procedure of its own: it simply uses the system's built-in window procedure, **DefWindowProc**, for all of its message processing. For an example of a real, live window procedure, we'll have to turn to our fully developed example program, WiniEdit.

Window Procedures

The real heart of a Windows program is its *window procedure* (commonly known as a "winproc"), which analyzes and responds to the messages the program receives from the system. Just to review some of what we've already learned:

- Windows sends messages to a window.
- The window belongs to a window class.
- The window class has a window procedure, identified as a parameter when the class was registered.
- All messages sent to the window are passed to its class's window procedure for processing.

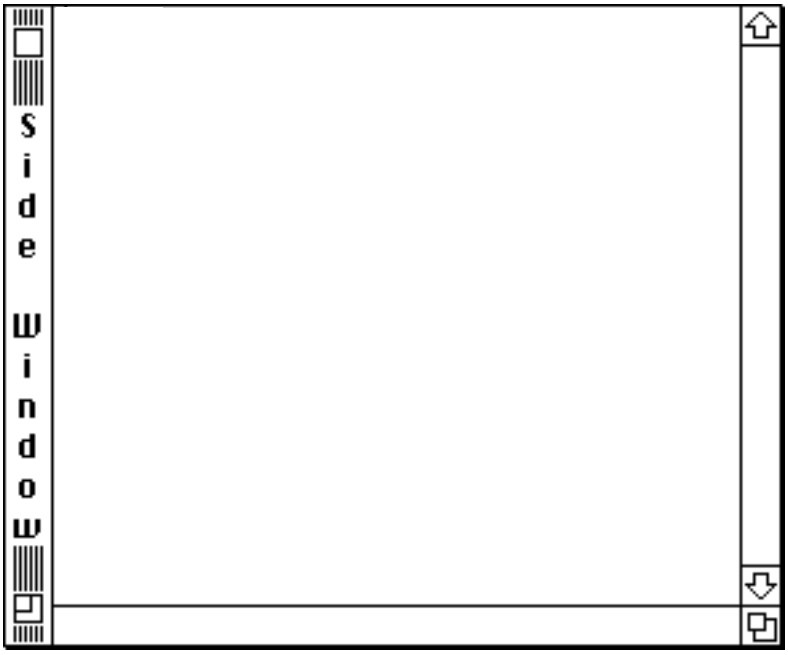
Theoretically, a program could register more than one window class and thus define more than one window procedure; but in practice, the great majority of Windows programs have just one window procedure that receives all messages addressed to any of their windows.

One way to think about the window procedure is to compare it to the Macintosh window definition function, or **`WDEF`**. Both accept messages sent by the system at strategic times to allow you to customize some aspects of a window's operations. But the Macintosh version accepts only a very limited set of messages and no others (Table 3–4). This means the Macintosh **`WDEF`** is only screen-deep: it lets you change the window's superficial appearance, but not its underlying behavior. On the other hand, since it's a resource, it can be incorporated into any program's application file with a tool like ResEdit, without touching the code of the program itself. In Volume Three of my *Macintosh Revealed* series, for instance, I developed a sample window definition function named SideWindow that displays a window with its title bar running vertically down the side instead of horizontally across the top (Figure 3–2). Simply by compiling this function into a **`'WDEF'`** resource and copying it into my program's application file with ResEdit, I was able to produce a version of MiniEdit that displayed its windows in this form.

Table 3–4. Macintosh window messages

Message code	Meaning
`wNew`	Initialize new window
`wCalcRgns`	Calculate structure and content regions
`wDraw`	Draw window frame
`wDrawGIcon`	Draw size region
`wGrow`	Draw feedback image for resizing window
`wHit`	Find where mouse was pressed
`wDispose`	Prepare to dispose of window

Figure 3–2. A Macintosh side window



By contrast, the Windows window procedure is part of the program itself, so it can't be changed without recompiling the entire program. It can have any name the program chooses to give it (in WiniEdit, it's named **DoMessage**), but must always have the following standard function signature:

```
LONG CALLBACK DoMessage (HWND   thisWindow,  // Handle to window receiving message
                         UINT   messageCode, // Message identifier
                         WPARAM wParam,      // Type-dependent information
                         LPARAM lParam)      // Type-dependent information
```

The keyword **CALLBACK** is required because the window procedure is a *callback function:* one that's designed to be called by the Windows system, rather than from within the program itself. Notice that the procedure's four parameters are the same as the first four fields of the message structure, which we looked at earlier (Listing 3–1). The parameters **wParam** and **lParam**, you'll recall, contain message-dependent information that varies from one type of message to another. (The names are a holdover from earlier versions of the Windows system, in which one of the parameters was a 16-bit word and the other a 32-bit long word; in Win32, both the **WPARAM** and **LPARAM** data types are equated to **LONG**, so both parameters are actually 32 bits.) The procedure returns a result of type **LONG**, whose significance is also message-dependent. The meanings of the parameters and result for any particular message type are given in the *Win32 Programmer's Reference.*

Listing 3–4 shows WiniEdit's window procedure. It's essentially just a big **switch** statement that examines the message type and calls the appropriate routine within the program to respond to it. The **switch** has a branch for each type of message that the program chooses to deal with explicitly. In WiniEdit's case, there are only ten of these: **WM_CREATE**, **WM_SIZE**, **WM_CTLCOLOREDIT**, **WM_SYSCOLORCHANGE**, **WM_SETFOCUS**, **WM_INITMENUPOPUP**, **WM_COMMAND**, **WM_QUERYENDSESSION**, **WM_CLOSE**, and **WM_DESTROY**. We'll be discussing

a few of these in the rest of this chapter and the others later in the book. All other messages fall through into the **`switch`** statement's d**`efault`** branch, which simply relays them to the system's default window procedure, **`DefWindowProc`**, to be dealt with in the standard way. Since many of the messages directed to a window are really designed to be handled by Windows itself, this last step is crucial to allow the system to receive these messages.

Listing 3–4. `WiniEdit`'s window procedure

```
LONG CALLBACK DoMessage (HWND thisWindow, UINT msgCode, WPARAM wParam, LPARAM lParam)

   // Get and process one message.

   {
      LONG  result = 0;                                    // Function result

      ErrorFlag = FALSE;                                   // Clear I/O error flag

      switch ( msgCode )                                   // Dispatch on message code
         {
            case WM_CREATE:
               DoCreate (thisWindow, wParam, lParam);          // Handle WM_CREATE message
               break;

            case WM_SIZE:
               DoSize (thisWindow, wParam, lParam);            // Handle WM_SIZE message
               break;

            case WM_CTLCOLOREDIT:
               result = DoCtlColorEdit (thisWindow, wParam, lParam); // Handle WM_CTLCOLOREDIT message
               break;

            case WM_SYSCOLORCHANGE:
               DoSysColorChange (thisWindow, wParam, lParam);  // Handle WM_SYSCOLORCHANGE message
               break;

            case WM_SETFOCUS:
               DoSetFocus (thisWindow, wParam, lParam);        // Handle WM_SETFOCUS message
               break;
```

Listing 3–4. `WiniEdit`'s window procedure (*continued*)

```
      case WM_INITMENUPOPUP:
         DoInitMenuPopup (thisWindow, wParam, lParam);   // Handle WM_INITMENUPOPUP message
         break;

      case WM_COMMAND:
         DoCommand (thisWindow, wParam, lParam);         // Handle WM_COMMAND message
         break;

      case WM_QUERYENDSESSION:
         result = DoQuery (thisWindow, wParam, lParam);  // Handle WM_QUERYENDSESSION message
         break;

      case WM_CLOSE:
         DoClose ();                                     // Handle WM_CLOSE message
         break;

      case WM_DESTROY:
         DoDestroy (thisWindow, wParam, lParam);         // Handle WM_DESTROY message
         break;
      default:
         result = DefWindowProc (thisWindow, msgCode,    // Pass message to Windows
                                 wParam, lParam);        //    for default processing
         break;

   }  /* end switch ( msgCode ) */


return result;


}  /* end DoMessage */
```

Program Control Messages

Table 3–5 lists a few messages related to overall program control. We've already mentioned the most important of these, **`WM_QUIT`**, several times. Windows sends the **`WM_QUIT`** message when a program asks for it by calling the Windows function **`PostQuitMessage`**. This is the only message that causes the **`GetMessage`** function to return a **`FALSE`** result, signaling the program to fall out of its message loop and terminate.

The other two messages in the table are invoked when the user attempts to end the Windows session by exiting from Windows or shutting down the computer. Before shutting down, Windows sends a **`WM_QUERYENDSESSION`** message to each running program, asking the program's permission to end the session. If a program is in an inconvenient state, such as in the middle of an operation that shouldn't be interrupted, it can deny permission by returning a **`FALSE`** result to this message. Listing 3–5, for instance, shows how WiniEdit handles the **`WM_QUERYENDSESSION`** message. (Portions of the code are shown in skeleton form because they involve parts of the Windows programming interface that we haven't discussed yet.) The

Table 3–5. Messages relating to program control

Message type	Meaning
WM_QUIT	Terminate program
WM_QUERYENDSESSION	OK to end Windows session?
WM_ENDSESSION	Windows session ending

WiniEdit function for responding to this message, **DoQuery**, in turn calls another program function, **CloseDoc**, to close the document displayed in its document window. If the contents of the doucment are "dirty" (have been changed since being read from or written to the disk), **CloseDoc** displays a message box asking the user whether to save the document before shutting down. If the user selects the "Yes" button, the function saves the window's contents to the the disk and returns a boolean result of **TRUE**; **DoQuery** in turn relays this value back to the system, indicating permission to proceed with the system shutdown. If the user selects "No," or if the document isn't dirty, it returns the same result, but without saving the document. (In each of these cases, **CloseDoc** also takes care of some other housekeeping, closing the document file and clearing the text displayed in the window to empty.) However, if the user selects the third choice in the message box, "Cancel" (or if an error occurs in saving the document to the disk), the function returns a **FALSE** result, denying permission for the system to shut down.

As soon as any program returns a **FALSE** response to the **WM_QUERYENDSESSION** message, Windows immediately stops sending such messages to the remaining programs and cancels the system shutdown; if, on the other hand, every running program responds **TRUE**, the system proceeds to shut down as planned. In either case, it sends each program a **WM_ENDSESSION** message with a boolean parameter indicating whether the shutdown is proceeding. If the parameter value is **TRUE**, this message constitutes a "goodbye kiss," notifying the program that the system is definitely about to shut down and giving it one last chance to do any final housekeeping it may require. If the parameter is **FALSE**, the message informs the program that the intended system shutdown, announced by the previous **WM_QUERYENDSESSION** message, has been canceled.

Listing 3–5. Handle WM_QUERYENDSESSION message

```
BOOL DoQuery (HWND thisWindow, UINT wParam, LONG lParam)

   // Handle WM_QUERYENDSESSION message.

   {
      BOOL  confirmed;                  // Did user confirm operation?


      confirmed = CloseDoc ();          // Allow user to save document if necessary
      return confirmed;                 // Report confirmation or cancellation

   }  /* end DoQuery */
```

Listing 3–5. Handle `WM_QUERYENDSESSION` message (*continued*)

```
BOOL CloseDoc (VOID)

   // Close document displayed in window.

   {
      BOOL  dirty;                        // Window contents changed since last save?
      INT   msgResult;                    // Result value returned by message box
      BOOL  confirmed;                    // Did user confirm operation?


      dirty = /* Has text been edited? */;
      if ( dirty )
         {
            msgResult = /* Display message box on screen */;

            switch ( msgResult )               // Dispatch on message result
               {
                  case IDYES:
                     /* Save window contents to disk */;
                     confirmed = !ErrorFlag;    // Confirm if no error
                     break;

                  case IDNO:
                     confirmed = TRUE;          // Confirm without saving
                     break;

                  case IDCANCEL:
                     confirmed = FALSE;         // Cancel operation
                     break;

               }  /* end switch ( msgResult ) */

         }  /* end if ( dirty ) */

      else
         confirmed = TRUE;                     // Confirm if not dirty


      if ( confirmed )                         // Did user confirm operation?
         {
            if ( (TheFile != NULL) )           // Is window associated with a file?
               /* Close file */

            /* Clear edit control's text */

         }  /* end if ( confirmed ) */


      return confirmed;                        // Report confirmation or cancellation


   }  /* end CloseDoc */
```

Sending Messages to a Window

Sometimes, in the course of your program's operations, it's convenient to be able to send a message to a window yourself—either to one of your own windows or to one belonging to another program or process. There are two ways of doing this, depending on whether you want the message sent in queued or unqueued form. The Windows function **PostMessage** is analogous to **PostEvent** on the Macintosh; it places a specified message in a window's message queue (more precisely, in the queue for the thread that created the window). It then returns control immediately to the point of call, without waiting for the message to be retrieved and processed by the thread's message loop. Note that you can post any message you like this way: it doesn't have to be one of the normal queued messages listed earlier in Table 3–2.

The second function, **SendMessage**, dispatches a message directly to the relevant window procedure. In this case, the processing of the message takes place synchronously: by the time control returns from the **SendMessage** call, the window procedure will already have processed the message to completion.

As on the Macintosh, you can define message types of your own and use **PostMessage** or **SendMessage** for internal coordination within your program itself. The Macintosh Event Manager originally reserved four of the sixteen possible event types for application use, though one of them has since been reclaimed by the system. In Windows, you can use any message type from **0x0400** to **0x7FFF** for your own purposes. The Windows API header files define a constant named **WM_USER** denoting the lower limit of this range (**0X0400**). Such messages are valid only for sending private messages from one window to another within the same window class (that is, under the control of the same window procedure). For communication between windows of different classes (those belonging to different application programs, for instance), you have to obtain an officially registered message type from the Windows function **RegisterWindowMessage**. Message types in the range **0xC000** to **0xFFFF a**re reserved for messages in this category.

Message Operations

Table 3–6 summarizes some common Windows functions relating to messages and message queues. We've already discussed some of them in this chapter; you can learn about the rest in the *Win32 Programmer's Reference.*

Table 3–6. Common message functions

Function	Mac counterpart	Purpose
`GetMessage`	`WaitNextEvent`	Retrieve next message from queue or yield control until one arrives
`PeekMessage`	`GetNextEvent`	Retrieve next message from queue, if any
`WaitMessage`	————	Yield control until message arrives in queue
`GetQueueStatus`	`EventAvail`	Check queue for presence of selected messages
`GetInputState`	————	Any mouse or keyboard messages pending in queue?
`TranslateMessage`	————	Translate raw keyboard message to character equivalent
`DispatchMessage`	————	Dispatch message retrieve from queue to window procedure
`GetMessagePos`	`EventRecord.where`	Mouse position at time of message
`GetMessageTime`	`EventRecord.when`	Time message was posted
`PostMessage`	`PostEvent`	Place message in queue
`SendMessage`	`PostEvent`	Send unqueued message directly to window procedure
`RegisterWindowMessage`	————	Define message type for interapplication communication
`PostQuitMessage`	————	Signal program completion

The Same Thing...

- The Macintosh Toolbox communicates with programs by sending them events.
- Macintosh programs are based on an event loop that retrieves and processes events one at a time.
- A Macintosh program receives the user's mouse clicks and keystrokes in the form of events.
- A Macintosh event is represented by an event record.
- A Macintosh event record has fields giving the type of event, the time it was posted, and the coordinates of the mouse at the time.
- Each Macintosh program has an event queue in which the system posts events and the program retrieves them with **`GetNextEvent`** or **`WaitNextEvent`**.

- Windows communicates with programs by sending them messages.
- Windows programs are based on a message loop that retrieves and processes messages one at a time.
- A Windows program receives the user's mouse clicks and keystrokes in the form of messages.
- A Windows message is represented by a message structure.
- A Windows message structure has fields giving the type of message, the time it was posted, and the coordinates of the mouse at the time.
- Each Windows program (or each thread within a multithreaded program) has a message queue in which the system posts messages and the program retrieves them with **`GetMessage`**.

...Only Different

- Macintosh events are directed to the program as a whole.
- A Macintosh program's event loop processes each event directly by examining its type and transferring control to the appropriate part of the program to respond to it.
- The Macintosh has sixteen event types.

- Windows messages are directed to a particular window within the program.
- A Windows program's message loop passes each message to the Windows function **`DispatchMessage`**, which in turn dispatches the message back to the program's window procedure; the window procedure then examines the message type and transfers control to the appropriate part of the program to respond to it.
- Windows has hundreds of message types.

• Most Macintosh events report user-level actions such as mouse clicks and keystrokes.	• Most Windows messages report higher-level actions such as moving, resizing, or scrolling a window or sending it a menu command.
• Macintosh programs must explicitly respond to every event themselves.	• Windows provides a default window procedure that defines a standard response to each message; programs needn't handle an event explicitly unless their response differs from the default.
• Macintosh events have a single type-dependent parameter.	• Windows messages have two type-dependent parameters, one nominally word-length and the other a long word; in Win32, both are actually long words.
• Conceptually, all Macintosh events are posted to the event queue and retrieved with **`GetNextEvent`** or **`WaitNextEvent`**.	• Many Windows messages are not posted to the message queue, but sent directly to the relevant window procedure.
• A Macintosh program has to hit-test and decode its own mouse clicks (with help from the Toolbox) and call the appropriate Toolbox routines to respond to them.	• A Windows program can let the default window procedure hit-test and decode its mouse clicks, converting them into high-level messages representing the meaning of the click.

Chapter 4

What's in a Window?

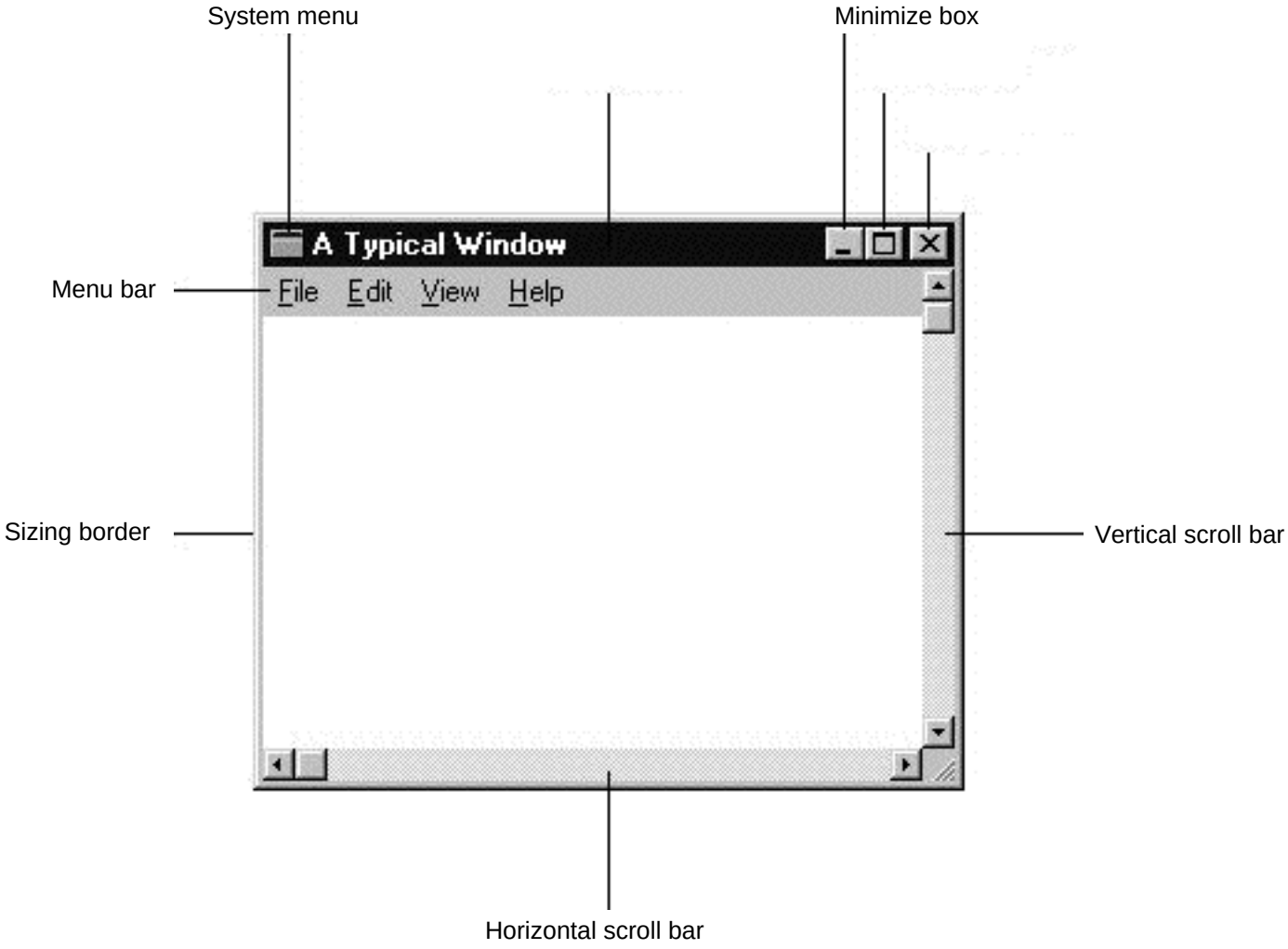
Because a program's windows are its only link with both the user and the system, a Windows application program normally has to maintain at least one window on the screen at all times, called its *main window,* and possibly others as well. In the last chapter, we learned how windows fit into the overall Windows programming model; in this chapter, we'll learn more about their properties and the operations a program can apply to them.

Anatomy of a Window

The Windows equivalent of a Macintosh document window is an *overlapped window,* so called because it can cover or be covered by other windows on the screen. The user can promote the window to the front by clicking it with the mouse or demote it by bringing some other window forward in front of it. Figure 4–1 shows a typical overlapped window in the Windows 95 user interface style. As you can see, it has most of the same components as a Macintosh window, though some of them are in different places or work in slightly different ways:

- At the top of the window is the *title bar* (also called a "caption bar"). Just as on the Macintosh, the user can grab the window by its title bar and drag it to a new position on the screen.
- The box with the little "x" at the right end of the title bar is the window's *close box.* Clicking in the close box closes the window, again just as on the Macintosh.
- Next to the close box is the *maximize box,* analogous to the Macintosh zoom box. Clicking here *maximizes* the window so that it takes up the entire working area of the screen. When the window is maximized, its maximize box changes into a *restore box* that restores it to its original size and position.
- Next to the maximize box is the *minimize box,* which reduces the window to iconic form. In older versions of the Windows user interface, a minimized window appeared as an icon at the bottom of the screen with a title beneath it, similar to a file or folder icon in the Macintosh Finder. In Windows 95, a minimized window becomes a button in the *task bar* along one edge of the screen. Clicking this button activates the window, restoring it to its original size and position.
- The small icon at the left end of the title bar represents the *system menu,* rather like the Apple symbol at the left end of the Macintosh menu bar. Clicking on the icon causes the menu to appear. Instead of a list of available desk accessories (which don't exist in Windows), the system menu offers commands for manipulating the window itself on the screen (see Figure 4–2). A program can add commands of its own to this menu, but the standard commands

Figure 4–1. Anatomy of a window



merely duplicate the functions of other components of the window (`Move`, `Size`, `Maximize`, `Minimize`, `Restore`, `Close`), giving the user an alternative way of accomplishing these same tasks.

- In Windows, each window has its own *menu bar* directly below the title bar, rather than a single menu bar at the top of the screen as on the Macintosh. In Windows terminology, the menu bar itself is considered a menu, the window's *main menu;* the individual menus that pop up when you click their titles are *submenus.* Another difference is that it isn't necessary to hold down the mouse button to keep a Windows menu visible on the screen: the menu appears when its title is clicked and remains visible even after the button is released. Clicking again in one of the menu's items selects that item and dismisses the menu; clicking outside the menu dismisses it without selecting an item.
- Instead of a size box at the bottom-right corner, a *sizing border* runs around all four sides of the window. The user can grab the border at any edge or corner and resize the window in any direction.

Figure 4–2. The system menu

- The window's horizontal and vertical scroll bars are essentially the same as their Macintosh counterparts in both appearance and behavior.

The fixed structural components of a window, called the window frame on the Macintosh, are known in Windows as the *nonclient area;* the interior of the window, which the Macintosh calls the content region, is the *client area.* In general, Windows takes responsibility for drawing and managing the nonclient area; what goes in the client area is your program's responsibility. Note that Windows considers the scroll bars part of the nonclient area, not part of the client area (content region) as on the Macintosh.

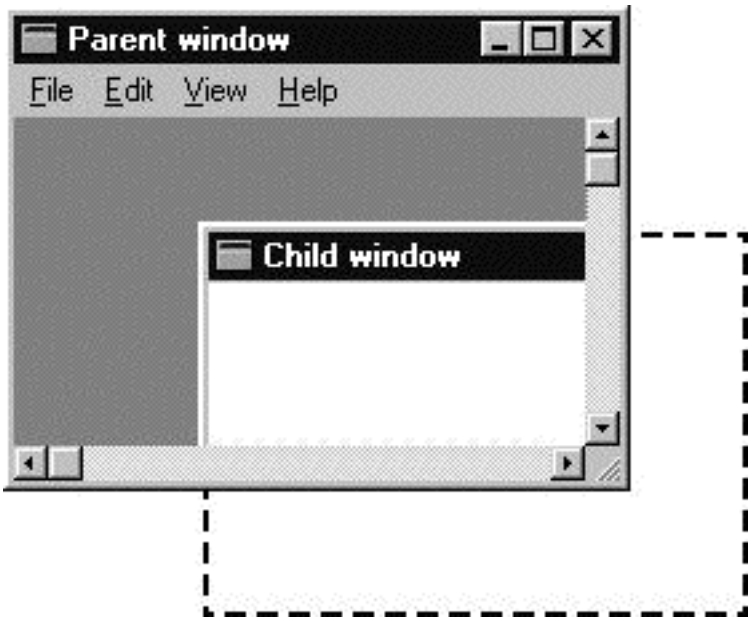
Child Windows

In Windows, one window can be a *child* of another window (called, logically enough, its *parent*). The child window is displayed within the client area of its parent, and is confined by the parent's border. Any part of the child window that extends past the edge of the parent window is clipped, so that only the portion within the parent's client area remains visible (see Figure 4–3). You can use child windows, for instance, to divide your main window into "panes" with different functions or purposes.

Every child window has an identifying number, assigned by the application program, called its *child-window identifier.* The child window uses this value to identify itself when communicating with its parent. A child window may in turn have children of its own, forming a hierarchy of ancestors and descendants like a family tree. Windows at the same level of the hierarchy, sharing the same parent, are called *sibling windows.* The window at the root of the tree, with no parent of its own, is called a *top-level window.* Every program has at least one top-level window, and may in fact have more than one.

Probably the most common examples of child windows are controls, such as pushbuttons, checkboxes, and so forth. On the Macintosh, controls are a separate kind of entity, different from windows; every control belongs to some window and appears within that window's content region. In Windows, a control is simply another kind of window, a child of the window in which it appears. We'll be discussing controls in Chapter 7.

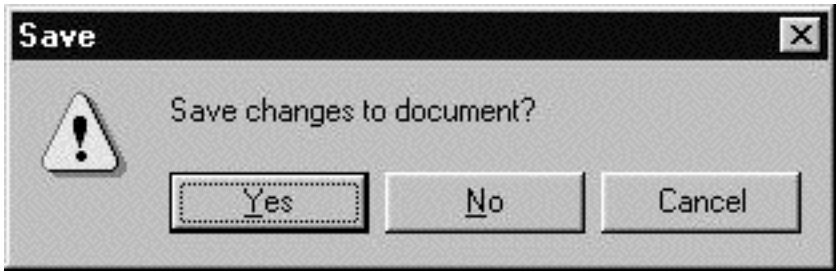
Figure 4–3. A child window clipped to its parent



Pop-Up Windows

One special kind of overlapped window is a *pop-up window*, such as a dialog or message box, that appears on the screen in response to some particular condition or action by the user. Pop-up windows typically have a title bar, system menu, and close box, but no sizing border, menu bar, scroll bars, or maximize and minimize boxes (see Figure 4–4). Normally, a pop-up window is associated with some other window, its *owner*, to which it reports the results of its interactions with the user. The pop-up always appears in front of its owner on the screen; if the user minimizes the owner, the pop-up becomes hidden, and reappears when the owner is restored to normal size. If the user closes the owning window, the pop-up is destroyed along with it. Notice that this ownership relationship is not the same as the parent–child relationship described in the preceding section.

Figure 4–4. A pop-up window



Z Order

In addition to its horizontal and vertical (X and Y) position on the screen, every window has a position in the third dimension, from front to back as viewed by the user on the screen. This front-to-back ordering is called the window's *Z order*. The window at the top of the Z order appears frontmost on the screen, partially or completely obscuring any other windows behind it. At the very bottom of the Z order is the *desktop window*, which serves as the background against which all other windows are displayed (analogous to the Window Manager port on the Macintosh).

The system automatically maintains the Z order as the user manipulates windows on the screen. Only top-level windows (those without a parent) belong to the main Z order; the children of a given parent window have a separate Z order of their own, which determines their appearance in case they overlap within the parent's client area. The frontmost window on the screen is the *active window*, and is displayed with a specially highlighted border and title bar. Only a top-level window can be the active window; if a child window is frontmost on the screen, its parent (or top-level ancestor) is considered the active window.

At the time of its creation, a window can be designated as a *topmost* window (not the same as a *top-level* window). A topmost window always appears ahead of all non-topmost windows in the Z order; it can be overlapped only by another topmost window. This property is typically used for message boxes that are meant to attract the user's attention and remain frontmost on the screen until dismissed.

Creating and Destroying Windows

The Windows function for creating a window is called, naturally enough, **`CreateWindow`**; Listing 4–1 shows its definition. The function creates a window with a specified class and other parameters and returns a window handle of type **`HWND`**. As we learned in Chapter 3, a program typically uses its own name as the name of its main window class. The Windows system also has a number of predefined classes for standard window types, with names like **`"BUTTON"`**, **`"LISTBOX"`**, and **`"SCROLLBAR"`**.

Just as in registering a class, when you create a window you supply a collection of style flags as a parameter, specifying various properties and characteristics of the window to be created. Those that apply to all windows, regardless of type, begin with the prefix **`WS_`**, for "window style," and are listed in Table 4–1. Some of the built-in window classes have additional style flags that are specific to those window types only, with prefixes such as **`BS_`** for "button style," **`LBS_`** for "list-box style," **`SBS_`** for "scroll-bar style," and so on. We'll discuss these class-specific styles when we cover the window types they refer to.

Like the window-class style options that we looked in Chapter 3, the window options in Table 4–1 denote bit masks that you can combine with the bitwise logical operators **`&`**, **`|`**, **`^`**, and **`~`** to form any combination of styles you need. A few of the options are mutually dependent: for instance, a window with the **`WS_SYSMENU`** style won't actually display its system menu on the screen unless it also has the **`WS_CAPTION`** style, giving it a title bar to display the menu in. Other options are mutually exclusive: a window can't have both a sizing border (**`WS_THICKFRAME`**) and a nonsizing border (**`WS_BORDER`**), or be both maximized (**`WS_MAXIMIZE`**) and minimized (**`WS_MINIMIZE`**); a window can't be both a pop-up (**`WS_POPUP`**) and a child of some other

Listing 4–1. Definition of **CreateWindow**

```
HWND CreateWindow
   {
      LPCTSTR  lpszClassName;          // Pointer to class name
      LPCTSTR  lpszWindowName;         // Pointer to window's title
      DWORD    dwStyle;                // Style options
      int      x;                      // Horizontal position of top-left corner
      int      y;                      // Vertical position of top-left corner
      int      nWidth;                 // Width of window in screen units
      int      nHeight;                // Height of window in screen units
      HWND     hwndParent;             // Handle to parent or owner
      HMENU    hmenu;                  // Handle to menu or child-window identifier
      HANDLE   hinst;                  // Handle to program instance
      LPVOID   lpvParam;               // Pointer to additional initialization data
   };
```

window (**WS_CHILD**). Finally, some of the styles listed in the table are simply shorthand abbreviations for the most common combinations of other styles, like **WS_OVERLAPPEDWINDOW** (a combination of **WS_OVERLAPPED**, **WS_CAPTION**, **WS_SYSMENU**, **WS_THICKFRAME**, **WS_MAXIMIZEBOX**, and **WS_MINIMIZEBOX**) or **WS_POPUPWINDOW** (a combination of **WS_POPUP**, **WS_BORDER**, and **WS_SYSMENU**).

> There's also an extended form of the **CreateWindow** function, called **CreateWindowEx**, that provides for more style options than the plain **CreateWindow** form. It does this by accepting an additional style parameter, a double word (**DWORD**) named **dwExStyle**. The extra parameter is the function's *first* parameter, preceding all of those shown in Listing 4–1. The additional options available through this parameter are the ones shown at the end of Table 4–1, with the prefix **WS_EX_** instead of just **WS_**.

The next four parameters to the **CreateWindow** function give the initial position and size of the new window on the screen. Specifying these values is complicated by a problem that we in the Macintosh world have never had to deal with: the variety of different graphical devices that Windows users can connect to their machines. As a Macintosh partisan, I've made it a point over the years to know as little as possible about MDAs, CGAs, EGAs, VGAs, and all the rest of those mysterious beasts, and I don't really want to start learning about them now. All those alphabet-soup graphics adapters have different dot resolutions (total number of dots on the screen horizontally and vertically), dot pitches (density of dots per linear inch), and aspect ratios (number of horizontal dots corresponding to each vertical dot). The parameters that specify a window's location and size are expressed in device-dependent units that vary with these metric characteristics of the screen. The Windows Graphical Device Interface allows you to obtain the device's metric characteristics and take them into account in positioning your windows, but we won't be discussing these matters until Chapter 6. Luckily, the **CreateWindow** function allows you to duck the whole issue by letting the system position your windows for you. If you specify the constant **CW_USEDEFAULT** for the window's coordinates and dimensions, Windows will automatically choose a reasonable position and size for it, based on the characteristics of the available graphics device. Once the window is on the screen, of course, it can be freely repositioned or resized wherever the user likes.

Table 4–1. Window style options

Style name	Meaning
WS_OVERLAPPED	Normal overlapped window
WS_OVERLAPPEDWINDOW	Normal overlapped window with title bar, system menu, sizing border, maximize and minimize boxes
WS_POPUP	Dialog-style pop-up window
WS_POPUPWINDOW	Dialog-style pop-up window with nonsizing border and system menu (visible only if window also has a title bar)
WS_CHILD	Child of another window
WS_CAPTION	Has title bar
WS_SYSMENU	Has system menu (visible only if window also has a title bar)
WS_MAXIMIZEBOX	Has maximize box
WS_MINIMIZEBOX	Has minimize box
WS_VSCROLL	Has vertical scroll bar
WS_HSCROLL	Has horizontal scroll bar
WS_THICKFRAME	Has sizing border
WS_BORDER	Has nonsizing border
WS_DLGFRAME	Has dialog-style double border and no title bar
WS_VISIBLE	Initially visible on screen
WS_MAXIMIZE	Initially maximized
WS_MINIMIZE	Initially minimized
WS_DISABLED	Initially disabled (doesn’t receive user input)
WS_TABSTOP	Can receive input focus via Tab key
WS_CLIPCHILDREN	Don’t overwrite children when drawing in this window
WS_CLIPSIBLINGS	Don’t overwrite overlapping siblings
WS_GROUP	First of a group of controls
WS_EX_DLGMODALFRAME	Has dialog-style double border with title bar
WS_EX_TOPMOST	Appears in front of all except other topmost windows
WS_EX_TRANSPARENT	Doesn’t obscure windows behind it
WS_EX_ACCEPTFILES	Accepts drag-and-drop files
WS_EX_NOPARENTNOTIFY	Don’t notify parent when created or destroyed (child windows only)

When you create a child or pop-up window, you supply a handle to the window’s parent or owner; if you’re creating a stand-alone, top-level window (such as your program’s main window), you can just leave this parameter **NULL**. For an overlapped or pop-up window, you can also supply a handle to a top-level menu to be displayed in the window’s menu bar. More often than not, though, you’ll set this parameter to **NULL**, calling for the default menu associated with the window’s class. For child windows (which have no menus of their own), the menu handle is

replaced by the integer child-window identifier that the window uses in communicating with its parent.

After creating a new window, the **CreateWindow** function sends a message (**WM_CREATE**) to the window's window procedure, telling it to do any initialization it wishes to perform to get itself ready for operation. The last parameter you supply when calling **CreateWindow** is a pointer to an optional data structure containing any additional information your window procedure may need in performing its initialization. **CreateWindow** simply relays this parameter to the window procedure as a parameter to the initialization message; its contents and interpretation are up to you to define. If you have no additional initialization data to pass to your window procedure, you can simply pass **NULL** for this parameter.

When you no longer need a window, you can get rid of it with the Windows function **DestroyWindow**. This releases any internal data associated with the window and makes the storage space it occupies available for reuse. Needless to say, any handles you may still be holding to the window become invalid and must not be used again.

Window and Class Attributes

As we learned in Chapter 2, you can't use a Windows handle to "open up" the underlying data structure and look inside at its internal fields. Instead, Windows provides functions that you can call to request various attributes of the object. In the case of a window, the function you use is **GetWindowLong**. You provide a handle to the window, along with an integer selector that identifies the particular attribute you want. For instance, to get a window's style flags, you could use a statement like

```
windowStyle = GetWindowLong(theWindow, GWL_STYLE);
```

Similarly, if you wanted to change the style flags, you would use the complementary function **SetWindowLong**. As an extra convenience, **SetWindowLong** returns the previous value of the specified attribute after changing it to the new value:

```
oldStyle = SetWindowLong (theWindow, GWL_STYLE, newStyle);
```

The selector **GWL_STYLE** used in the example is a constant defined in the Windows API header files; see Table 4–2 for a list of all the available selectors. In earlier versions of the Windows system, some of these application-accessible fields were single words (16 bits) and others were long words (32 bits). Accordingly, the programming interface provided two pairs of access functions: **GetWindowWord** and **SetWindowWord** for the 16-bit fields, **GetWindowLong** and **SetWindowLong** for those of 32 bits. In Win32, all of these fields are 32 bits long and are accessed with the long-word versions of the functions.

Table 4–2. `GetWindowLong` selectors

Selector	Meaning
`GWL_STYLE`	Style options
`GWL_EXSTYLE`	Extended style options
`GWL_WNDPROC`	Pointer to window procedure
`GWL_HINSTANCE`	Handle to program instance
`GWL_HWNDPARENT`	Handle to parent window (child windows only)
`GWL_ID`	Child-window identifier (child windows only)
`GWL_USERDATA`	Application-defined data

A similar pair of functions, **`GetClassLong`** and **`SetClassLong`**, access the fields of the window class rather than the window itself, using the selectors listed in Table 4–3. (Again, some of these fields were once 16 bits long instead of 32 and were accessed with the functions **`GetClassWord`** and **`SetClassWord`**; in Win32 they're all 32 bits and use **`GetClassLong`** and **`SetClassLong`** instead.) There's also a function named **`GetClassInfo`** that retrieves all of the data items shown in the table in one operation. This function accepts a pointer to a structure of type **`WNDCLASS`** (shown in Listing 3–3 of the last chapter) and fills it in with the same parameter values that were specified in registering the class originally.

A few of the data items associated with a window have retrieval functions of their own instead of being accessed with **`GetWindowLong`**. **`GetWindowText`** returns a window's title; **`SetWindowText`** changes it. Both of these functions accept a pointer to a string buffer and copy the title to or from the buffer; to find out how big a buffer you need, you can first call **`GetWindowTextLength`** to get the length of the title in characters. **`GetWindowRect`** takes a pointer to a rectangle structure and fills it in with the coordinates of the window's bounding rectangle, in screen coordinates. **`GetClassName`** accepts a handle to a window and returns the name of the window class it belongs to.

Another useful function is **`GetWindow`**, which returns a handle to another window related in some way to a given window. A selector parameter identifies the desired relationship. For example, if **`thisWindow`** is a handle to a pop-up window, the statement

Table 4–3. `GetClassLong` selectors

Selector	Meaning
`GCL_STYLE`	Style options
`GCL_WNDPROC`	Pointer to window procedure
`GCL_HMODULE`	Program instance that registered the class
`GCL_HICON`	Icon for minimizing windows
`GCL_HCURSOR`	Default cursor
`GCL_HBRBACKGROUND`	Brush for painting window background
`GCL_MENUNAME`	Resource name of menu
`GCL_CBCLSEXTRA`	Extra bytes of client data in class structure
`GCL_CBWNDEXTRA`	Extra bytes of client data in window structure

Table 4–4. `GetWindow` selectors

Selector	Meaning
`GW_CHILD`	Frontmost child (parent windows only)
`GW_OWNER`	Owning window (pop-up windows only)
`GW_HWNDFIRST`	Frontmost window like this one (top-level, child, or topmost)
`GW_HWNDLAST`	Rearmost window like this one (top-level, child, or topmost)
`GW_HWNDNEXT`	Window immediately behind this one
`GW_HWNDPREV`	Window immediately in front of this one

```
theOwner = GetWindow(thisWindow, GW_OWNER);
```

obtains a handle to its owner. Table 4–4 lists the possible selectors you can use with this function. The selectors **`GW_HWNDFIRST`**, **`GW_HWNDLAST`**, **`GW_HWNDNEXT`**, and **`GW_HWNDPREV`** all refer to a window of the same kind as the one you supply as a parameter: that is, if you pass a handle to a top-level window, you'll get back a handle to the first, last, next, or previous top-level window on the screen; if the window you supply is a topmost window, you'll get the first, last, next, or previous topmost window; if it's a child window, you'll get back its first, last, next, or previous sibling. You can get some of this same information more directly by calling **`GetTopWindow`** (like **`FrontWindow`** on the Macintosh) or **`GetNextWindow`** (which actually returns the next *or previous* window, depending on a parameter you supply).

Client Data

Application programs frequently need to associate some data of their own with the windows they create. For instance, you might want to record the path name of the document file displayed in each window, for use when the user asks to save the window's contents back to the file. Windows gives you several ways of attaching such additional *client data* to a window.

The simplest way is with the **`GWL_USERDATA`** selector to the **`SetWindowLong`** and **`GetWindowLong`** functions. This allows you to associate a single, 4-byte item of client data with each window, much like a Macintosh window's reference constant. As on the Macintosh, if you need more than 4 bytes of client data, you can allocate space for an auxiliary data structure in your heap and use the allotted 4 bytes to hold a pointer to it. For example, suppose you want to keep three items of data with each of your windows: an integer resource ID, a string pointer, and an integer length count for the string. You could define a data structure to hold them,

```
struct WindowData
   {
      SHORT  rsrcID;                          // Resource ID
      LPSTR  strPtr;                          // Pointer to string
      SHORT  strLength;                       // Length of string

   }; /* end WindowData */
```

create the structure in your heap,

```
theHeap  = GetProcessHeap();                          // Get handle to heap
dataSize = sizeof(WindowData);                        // Get block size
dataPtr  = HeapAlloc (theHeap, NULL, dataSize);       // Allocate the block
```

store the data into it,

```
dataPtr->rsrcID    = . . . ;                          // Set resource ID
dataPtr->strPtr    = . . . ;                          // Set string pointer
dataPtr->strLength = . . . ;                          // Set string length
```

and associate it with your window:

```
dataAddr = LONG(dataPtr);                             // Convert to long integer
SetWindowLong (theWindow, GWL_USERDATA, dataAddr);    // Associate with window
```

Alternatively, instead of allocating a block from the heap, you can put your auxiliary data right in the window record itself. You may recall, back in Listing 3–3, that the **WNDCLASS** parameter structure used in registering a class includes a pair of fields, **cbClsExtra** and **cbWndExtra**, which tell how many additional bytes of client data to allocate in the window class structure and the window structure, respectively. The additional data is accessible in the same way as the standard items, using **SetWindowLong** and **GetWindowLong** (or **SetClassLong** and **GetClassLong**) and their related word-length versions.

The selector parameter that you supply to these functions is actually a byte offset within the window or class structure. The standard selectors listed in Tables 4–2 and 4–3 all denote negative offsets; nonnegative values are reserved for client data. The first item of client data is at offset **0**, with succeeding items at increasing positive offsets, depending on the size of the items that precede them. So to associate the same three items with each window as in the example above, you might define the following offsets:

```
#define GWW_RSRCID     0                              // Resource ID       (SHORT)
#define GWL_STRPTR     GWW_RSRCID    + sizeof(SHORT)  // Pointer to string (LPSTR)
#define GWW_STRLENGTH  GWL_STRPTR    + sizeof(LPSTR)  // Length of string  (SHORT)
#define CLIENTBYTES    GWW_STRLENGTH + sizeof(SHORT)  // Total bytes of client data
```

You could then set the number of bytes of client data when you register the class

```
. . . ;
windowClass.cbWndExtra = CLIENTBYTES;                 // Set length of client data
. . . ;
RegisterClass (&windowClass);                         // Register the class
```

and, after creating a window belonging to the class, store your client data into the window record this way:

```
SetWindowWord (theWindow, GWW_RSRCID, ... );          // Set resource ID
SetWindowLong (theWindow, GWL_STRPTR, ... );          // Set string pointer
SetWindowWord (theWindow, GWW_STRLENGTH, ... );       // Set string length
```

Note that you have to be careful to use **SetWindowWord** for 2-byte (16-bit) fields and **SetWindowLong** for those of 4 bytes (32 bits). To retrieve the data from the window record, you would naturally use **GetWindowWord** and **GetWindowLong**:

```
rsrcID    = GetWindowWord(theWindow, GWW_RSRCID);     // Get resource ID
strPtr    = GetWindowLong(theWindow, GWL_STRPTR);     // Get string pointer
strLength = GetWindowWord(theWindow, GWW_STRLENGTH);  // Get string length
```

Although it's a more unusual thing to do, you can also use this same technique to store client data into the window class instead of the individual window structure, using the **cbClsExtra** field of the **WNDCLASS** structure to specify the size of your data and the functions **SetClassWord**, **SetClassLong**, **GetClassWord**, and **GetClassLong** to access it.

There is yet a third way of associating client data with a window, which is probably the most general and flexible of all. Instead of storing anything directly into the window record, you can associate *properties* with your windows. Every window has a *property list* to which you can add as many properties as you need. Each property is identified by a name that you supply as a string. The value of the property is nominally a handle, but you can use typecasting to make it any 32-bit value you like. You can add properties freely to a window's property list with the Windows function **SetProp**

```
propHandle = HANDLE(strPtr);                        // Convert to handle
SetProp (theWindow, "Prop_StrPtr", propHandle);     // Store as property
```

and retrieve them with **GetProp**:

```
propHandle = GetProp(theWindow, "Prop_StrPtr");     // Get property
strPtr     = LPSTR(propHandle);                     // Convert from handle
```

Before destroying a window, you must destroy all of its properties. The Windows function **RemoveProp** removes a property from the window's property list:

```
RemoveProp (theWindow, "Prop_StrPtr");              // Remove property from window
```

If the value of the property is a pointer or handle to a block of memory in your heap, you should, of course, deallocate the block after removing the property.

Window Display

Simply creating a window doesn't automatically make it appear on the screen; you have to display it explicitly by calling **ShowWindow**. This is a general-purpose function for setting a window's *show state*, which specifies not only whether it is visible or invisible, but also whether it's active or inactive and whether it's maximized, minimized, or in its normal (restored) state. The parameter specifying the window's new show state can take any of the values shown in Table 4–5. Aspects of the show state that are not mentioned in the second column of the table are unaffected by the given parameter: for instance, the parameter **SW_SHOW** makes the window visible and active, but doesn't change whether it's maximized, minimized, or restored. Some of the parameters are simply alternative names for the same show state: **SW_SHOWNOACTIVATE** and **SW_SHOWNA** are equivalent, as are **SW_SHOWNORMAL** and **SW_RESTORE**.

Table 4–5. `ShowWindow` parameters

Parameter	Meaning
`SW_SHOW`	Visible, active
`SW_HIDE`	Invisible, inactive
`SW_SHOWNOACTIVATE`	Visible
`SW_SHOWNA`	Visible
`SW_MINIMIZE`	Minimized, inactive
`SW_SHOWMINIMIZED`	Minimized, visible, active
`SW_SHOWMINNOACTIVE`	Minimized, visible
`SW_SHOWMAXIMIZED`	Maximized, visible, active
`SW_RESTORE`	Normal size, visible, active
`SW_SHOWNORMAL`	Normal size, visible, active

Window-Related Messages

Table 4–6 lists a few of the messages relating to windows and window manipulation that a window can receive. As we've already seen, one of the first messages a window receives, right after it is created, is **`WM_CREATE`**. Programs can use this occasion to set up any data structures or perform any other initialization they may need to do to get the window in working order, such as creating child windows, initializing properties, and so forth. In WiniEdit's case, the main order of business is to create an *edit control*, a child window similar to a TextEdit record on the Macintosh, and associate it with the window to handle cut-and-paste text editing. We'll be talking about edit controls in Chapter 8, so we'll postpone looking at WiniEdit's **`DoCreate`** routine until then.

The matching bookend to the **`WM_CREATE`** message is **`WM_DESTROY`**, sent by the Windows function **`DestroyWindow`** to allow the program to clean up after itself when a window is about to be destroyed. Listing 4–2 (**`DoDestroy`**) shows how WiniEdit handles this message. There's no need to destroy the window's edit control explicitly, since the **`DestroyWindow`** function automatically destroys all of a window's children along with it (as well as its menu and any other windows it owns). Graphics objects like the window's background brush, however, must be explicitly destroyed. Since WiniEdit is a single-document application, destroying its one and only window is tantamount to exiting from the program, so its **`DoDestroy`** function calls the Windows function **`PostQuitMessage`**. As we discussed in Chapter 3, this posts a **`WM_QUIT`** message to the program's message queue; when the program's message loop eventually retrieves this message from the queue, its call to **`GetMessage`** will return a **`FALSE`** result, causing the message loop to end and the program to terminate.

Table 4–6. Common window-related messages

Message type	Meaning
WM_CREATE	Window being created; initialize its associated data structures
WM_SHOWWINDOW	Show or hide window on screen
WM_NCPAINT	Repaint nonclient area
WM_PAINT	Repaint client area (like Macintosh update event)
WM_ERASEBKGRND	Erase background of client area
WM_ACTIVATE	Window being activated or deactivated
WM_MOVE	Window has been moved
WM_SIZE	Window's size has changed
WM_VSCROLL	Scroll window's contents vertically
WM_HSCROLL	Scroll window's contents horizontally
WM_CLOSE	Close window
WM_DESTROY	Window being destroyed; deallocate or finalize its data structures

Important window messages between **WM_CREATE** and **WM_DESTROY** include **WM_SHOWWINDOW** when the window is displayed or hidden on the screen, **WM_ACTIVATE** when it comes forward as the active window or is demoted in favor of another window, **WM_MOVE** when it is repositioned on the screen, and **WM_SIZE** when its size is changed.

One window message especially worth noting is **WM_PAINT**, which is the Windows counterpart of a Macintosh update event. Like the Macintosh Window Manager, Windows maintains a *visible region* and an *update region* for each window. The visible region keeps track of what portion of each window is currently exposed to view on the screen. As the user opens, closes, moves, sizes, minimizes, and maximizes windows, new parts of a window may come into view that weren't exposed before. Instead of drawing these immediately, Windows simply signals that they need to be drawn by adding them into the window's *update region.* Eventually, the program's message loop will call **GetMessage** to ask for the next message from the message queue. Any other

Listing 4–2. Handle **WM_DESTROY** message

```
VOID DoDestroy (HWND thisWindow, UINT wParam, LONG lParam)

   // Handle WM_DESTROY message.

   {
      DeleteObject (BackgroundBrush);                // Destroy background brush

      PostQuitMessage (NO_ERROR);                    // Signal program completion

   }  /* end DoDestroy */
```

messages pending in the queue will be reported first, but if the queue is empty, **GetMessage** will scan through the program's (or thread's) windows looking for nonempty update regions. If it finds one, it will return a **WM_PAINT** message for that window. Because such messages are generated "on the fly" instead of being physically posted to the queue, they have lower priority than user-input messages like **WM_LBUTTONDOWN** or **WM_CHAR**.

All of this, of course, should have a very familiar ring to Macintosh programmers. In fact, the way Windows generates **WM_PAINT** messages is virtually identical to the way update events work on the Macintosh. So it should come as no surprise that the familiar Macintosh technique of "deferred drawing" carries over to Windows as well. When the contents of a window on the screen need to change for any reason, a well-written Windows program doesn't just redraw them on the spot. Instead, just as on the Macintosh, the approved practice is to add the affected area into the window's update region and wait for the **WM_PAINT** message to do the actual redrawing. A pair of utility functions named **InvalidateRect** and **InvalidateRgn** add a rectangular area or an arbitrarily shaped region, respectively, into a window's update region; the inverse functions **ValidateRect** and **ValidateRgn** do the opposite, removing a rectangle or a region from the update region so that it *won't* be redrawn with a **WM_PAINT** message. You can find out a window's current update region with **GetUpdateRgn**; a related function, **GetUpdateRect**, returns the update region's bounding box, the smallest rectangle completely enclosing the region.

Actually, Windows generates *two* messages when it finds a window with a nonempty update region: a **WM_NCPAINT** message to draw the needed portions of the window's nonclient area and a **WM_PAINT** message for the client area. Programs typically don't process the **WM_NCPAINT** message themselves, but simply pass it along to the default window procedure to draw the window's frame in the standard way. If you wanted to, though, you could intercept this message and alter or customize the window's appearance on the screen, just like a Macintosh **WDEF**. You could use this capability, for instance, to implement the Windows equivalent of my Macintosh "side window" of Figure 3–2.

When it comes to a window's client area, though, only the program itself knows what to draw. The default window procedure responds to **WM_PAINT** messages by simply clearing the window's update region without drawing anything. If a program wants anything actually drawn in the window, it has to do the drawing itself. Again, this process is very similar to the way it's done on the Macintosh. The drawing operations are enclosed between calls to a pair of matching Windows functions, **BeginPaint** and **EndPaint**, analogous to **BeginUpdate** and **EndUpdate** on the Macintosh:

```
HDC         theContext;                              // Device context for window
PAINTSTRUCT paintInfo;                               // Info structure for painting

theContext = BeginPaint (thisWindow, &paintInfo);    // Open device context
   ...thcContext...;                                 // Use it to draw into window
EndPaint (thisWindow, &paintInfo);                   // Close device context
```

BeginPaint accepts a handle to the window to be updated, along with a pointer to a *paint structure* of type **PAINTSTRUCT** (Listing 4–3). The function fills in the fields of the paint structure with descriptive information about the needed painting operation and returns a handle to a device context (the Windows counterpart of a QuickDraw graphics port) for drawing into the window. Just as on the Macintosh, the context's clipping region is set to the intersection of the window's visible and update regions, so that everything drawn into the context will be clipped to

Listing 4–3. Paint structure

```
typedef struct _PAINTSTRUCT
   {
      HDC   hdc;                    // Handle to device context
      BOOL  fErase;                 // Erase background?
      RECT  rcPaint;                // Smallest rectangle enclosing update region
      BOOL  fRestore;               // Reserved for system use
      BOOL  fIncUpdate;             // Reserved for system use
      BYTE  rgbReserved[32];        // Reserved for system use

   }  PAINTSTRUCT;
```

just the relevant region of the screen. The program can then proceed to use this device context to draw the contents of the window, using the information in the paint structure, if necessary, to tell it what and how to draw. When it's finished, it calls **EndPaint** to restore the window to its original state; the device context becomes invalid and cannot be used again. Note that the **BeginPaint** and **EndPaint** calls are necessary in order to clear the window's update region; without them, you will keep receiving the same **WM_PAINT** message indefinitely.

To prepare the way for drawing into the update region, **BeginPaint** sends a **WM_ERASEBKGND** message to your window procedure. If you've defined a default background brush when registering the window's class, you can just ignore this message and let the default window procedure handle it by painting the region with the background brush. Alternatively, you can leave the class's background brush undefined (**NULL**) and process the **WM_ERASEBKGND** message explicitly in your window procedure. If you don't do either (that is, if you pass the message to **DefWindowProc** without defining a background brush for the class), the **BeginPaint** function will set the **fErase** flag in the paint structure to **TRUE**, giving you a second chance to erase the background yourself before drawing into it.

> It's also possible to bypass the message loop and send an explicit **WM_PAINT** message directly to a window's window procedure. The Windows function **UpdateWindow** checks a window's update region and sends an unqueued **WM_PAINT** message to the window procedure if the region is nonempty. Another function, **RedrawWindow**, can validate or invalidate a specified rectangle or region and *then* issue an unqueued message. This function also accepts a flag-word parameter that gives you even finer control over the exact sequence and content of **WM_NCPAINT**, **WM_ERASEBKGND**, and **NC_PAINT** messages that the window receives; see the *Win32 Programmer's Reference* for details.

Other Window Operations

Table 4–7 summarizes the most common of Win32's window-related operations. We've already discussed some of them in this chapter; you can learn about the rest in the *Win32 Programmer's Reference.*

Table 4–7. Common window functions

Function	Mac counterpart	Purpose
`CreateWindow`	`NewWindow`	Create new window
`DestroyWindow`	`DisposeWindow`	Destroy window
`GetWindow`	————	Get window in specified relation to a given window
`GetParent`	`ControlRecord.contrlOwner`	Get parent window
`SetParent`	`ControlRecord.contrlOwner`	Set parent window
`IsChild`	————	Is this window a descendant of a given window?
`FindWindow`	————	Find window with a given class and title
`GetClassName`	————	Get name of window's class
`IsWindow`	————	Is this a handle to an existing window?
`GetDesktopWindow`	`GetWMgrPort`	Get desktop window
`GetWindowWord`	————	Get word-length attribute of window
`SetWindowWord`	————	Set word-length attribute of window
`GetWindowLong`	————	Get long-word attribute of window
`SetWindowLong`	————	Set long-word attribute of window
`GetClassWord`	————	Get word-length attribute of window class
`SetClassWord`	————	Set word-length attribute of window class
`GetClassLong`	————	Get long-word attribute of window class
`SetClassLong`	————	Set long-word attribute of window class
`GetWindowText`	`GetWTitle`	Get window title
`GetWindowTextLength`	————	Get length of window title
`SetWindowText`	`SetWTitle`	Set window title
`GetWindowRect`	`WindowRecord.strucRgn`	Get window's bounding rectangle
`GetClientRect`	`WindowRecord.contRgn`	Get window's client rectangle
`AdjustWindowRect`	————	Calculate bounding rectangle from client rectangle

Table 4–7. Common window functions (*continued*)

Function	Mac counterpart	Purpose
`WindowFromPoint`	`FindWindow`	Which window contains a given point?
`ChildWindowFromPoint`	`FindControl`	Which of a window's children contains a given point?
`MoveWindow`	`MoveWindow`, `SizeWindow`	Set window's position and size
`SetWindowPos`	`MoveWindow`, `SizeWindow`, `BringToFront`, `SendBehind`	Set window's position, size, and Z order
`GetTopWindow`	`FrontWindow`, `WindowRecord.controlList`	Get frontmost window on screen or frontmost child of a given window
`GetNextWindow`	`WindowRecord.nextWindow`	Get window immediately behind this one
`BringWindowToTop`	`BringToFront`	Promote window to front of Z order
`GetForegroundWindow`	`FrontWindow`	Get active window
`SetForegroundWindow`	`SelectWindow`	Activate window
`EnableWindow`	————	Enable or disable window for user input
`ShowWindow`	`ShowWindow`, `HideWindow`, `ShowHide`, `ZoomWindow`	Set window's show state
`SetWindowPlacement`	————	Set window's show state and screen positions (maximized, minimized, and restored)
`GetWindowPlacement`	————	Get window's show state and screen positions (maximized, minimized, and restored)
`IsWindowVisible`	`WindowRecord.visible`	Is window visible?
`IsZoomed`	————	Is window maximized?
`CloseWindow`	————	Minimize window
`GetProp`	————	Get window property
`SetProp`	————	Set window property
`RemoveProp`	————	Remove window property
`EnumWindows`	————	Apply a function to all top-level windows on the screen
`EnumChildWindows`	————	Apply a function to all children of a given window

The Same Thing...

- Macintosh windows have a title bar, a close box, and scroll bars.
- A Macintosh window has a frame and a content region.
- A Macintosh window has a plane that expresses its front-to-back position on the screen.
- The Macintosh has a Window Manager port in which it draws the desktop background behind all windows on the screen.
- A Macintosh window has a graphics port for drawing its contents.
- A Macintosh window has a background pattern for filling the background of its content region.
- A Macintosh window's reference constant is an optional 4-byte value containing any data the program wishes to associate with the window, such as a pointer or handle to an auxiliary data structure.

- Windows windows have a title bar, a close box, and scroll bars.
- A Windows window has a nonclient area and a client area.
- A Windows window has a Z order that expresses its front-to-back position on the screen.
- Windows has a desktop window in which it draws the desktop background behind all windows on the screen.
- A Windows window has a device context for drawing its contents.
- A Windows window has a background brush for filling the background of its client area.
- A Windows window's **`GWL_USERDATA`** attribute is an optional 4-byte value containing any data the program wishes to associate with the window, such as a pointer to an auxiliary data structure.

...Only Different

- Macintosh windows have a size box.
- Macintosh windows have a zoom box.
- On the Macintosh, the scroll bars are part of a window's content region.
- The Macintosh has one global menu bar at the top of the screen.

- Windows windows have a sizing border.
- Windows windows have a maximize box and a minimize box.
- In Windows, the scroll bars are part of a window's nonclient area.
- In Windows, each window has its own menu bar.

- On the Macintosh, each window is an independent entity.
- On the Macintosh, controls, menus, and icons are separate kinds of entity, different from windows.
- A Macintosh program must supply its own code to respond to the user’s manipulations of window components such as the title bar, size box, or close box.
- When creating a new window, a Macintosh program must specify the window’s size and position on the screen.
- A Macintosh program can directly access the fields of a window record.
- The only client data in a Macintosh window record is the 4-byte reference constant.

- In Windows, one window can be the child, parent, or owner of another window.
- In Windows, controls, menus, and icons are kinds of window.
- A Windows program can rely on the default window procedure to provide standard responses to the user’s manipulations of window components.
- When creating a new window, a Windows program can let the system choose a default size and position on the screen.
- A Windows program can access the components of a window structure only indirectly, via Windows functions provided for the purpose.
- A Windows program can allocate any needed amount of client data in the window structure, and can also associate any number of properties with the window.

Chapter 5

Mouse and Keyboard

One of the most striking things about our example program, WiniEdit, is that it never has to deal directly with the user's mouse clicks and keystrokes. This is because it relies on the system's default window procedure, **`DefWindowProc`**, and the predefined window class **`EDIT`** to provide standard responses to the user's input actions: moving and manipulating windows on the screen, choosing menu items, selecting and typing text in the window's edit control, and so on. In the real world, of course, application programs have to be able to provide new functionality beyond what's already built into the system. This means intercepting the user's input actions and responding to them in the program's own way, according to the needs of its particular application. In this chapter, we'll learn how to receive and respond to input from the mouse and keyboard.

The Mouse

Like so many other things in the IBM/DOS world, the mouse has never been standardized. While Macintosh programmers could always count on a standard, stable one-button mouse, their DOS colleagues have had to deal with a proliferation of possibilities. The most common configuration is the two-button Microsoft mouse, but there are also systems in use with a one-button mouse, a three-button mouse, or no mouse at all. The Windows system is designed to support all of these possibilities, and Windows programs have to be prepared to accommodate any of them.

One thing this means in practical terms is always to provide a keyboard-based alternative to anything the user can do with the mouse. We'll see later how Windows supports this idea with alternative keyboard interfaces to such things as menus, controls, and scroll bars. In planning your own user interface, you should also give some thought to how your mouseless users can access your program's features from the keyboard alone.

Another factor to consider is how you'll deal with varying numbers of buttons on the mouse. By convention, the button on a one-button mouse is considered to be the left; those on a two-button mouse, the left and right. So one approach to button uncertainty is simply to ignore the right and middle buttons and provide a one-button interface based solely on the left button. This is certainly the easiest strategy for porting an existing Macintosh program to Windows. A variation on the same idea would be to treat all three buttons interchangeably, by intercepting all messages referring to the right and middle buttons and reposting them with **`SendMessage`** or **`PostMessage`** as equivalent left-button messages.

A more elaborate approach would be to use the right and middle buttons for convenience features that can also be accessed in other ways. For instance, many Windows programs use the right

mouse button to pop up a context-sensitive floating menu offering the most common or useful operations for a particular area of the screen. As long as all of the same commands are also available via the menu bar in the usual way, this "context menu" is simply a handy extra that you can offer to those users equipped to take advantage of it, without seriously penalizing those who aren't. Similarly, you might use the middle button as an alternative to Shift-click for extending an existing selection: again, nothing significant is lost, since users without a middle button can still perform the operation in another way. (Most Windows programs simply ignore the middle button entirely.)

If you design your programs this way, you don't normally need to know whether there's a mouse installed or how many buttons it has: you simply write your window procedure to handle clicks from the various mouse buttons *in case* you receive any. If you do need to find out what kind of mouse is available on the user's machine, you can use the Windows function **GetSystemMetrics**. This is a general-purpose function that can provide a whole range of information on the current system configuration—including, but not limited to, the mouse characteristics. The function takes a single parameter, an integer selector that identifies the particular item of information you want. All of the selectors are defined as interface constants beginning with the prefix **SM_**, for "system metrics." The call

```
hasMouse = GetSystemMetrics(SM_MOUSEPRESENT);
```

will return a boolean result telling whether the system has a mouse, and

```
nButtons = GetSystemMetrics(SM_CMOUSEBUTTONS);
```

will tell you how many buttons it has (or **0** if there is no mouse).

As a convenience to left-handed users, those with a two- or three-button mouse can use the Mouse control panel to swap the functions of the left and right buttons. There's a Windows function, **SwapMouseButton**, for doing this, but you shouldn't normally call it yourself, since it affects the global behavior of the system for all programs, not just your own. It's best simply to honor whatever preference the user has set through the control panel. If necessary, though, you can call **GetSystemMetrics** with the selector **SM_SWAPBUTTON**

```
isSwapped = GetSystemMetrics(SM_SWAPBUTTON);
```

to find out whether the buttons are currently swapped. Another property the user can set with the control panel is the mouse's tracking speed, which you can find out by calling the Windows function **SystemParametersInfo** with the selector **SPI_GETMOUSE**. The selector **SPI_SETMOUSE** lets you change the mouse speed, but again this is not something you should normally be doing from within an application program.

Mouse Messages

When the user moves the mouse or presses or releases one of its buttons, Windows reports this event to your program by sending one of the messages listed in Table 5–1. The message is normally sent to the window that contains the mouse at the time of the event. (There's an exception to this rule, which we'll learn about shortly.) So the first thing Windows has to do is figure out which window the mouse is in. It then sends the window a message of type **WN_NCHITTEST** to narrow down the specific part of the window where the event occurred (in the title bar, menu bar, sizing border, client area, or whatever). The correct handling of this very important message is critical to the proper functioning of the entire Windows user interface. It's

essential for your program to pass it through to the default window procedure, **`DefWindowProc`**, for processing.

The **lParam** parameter to **WM_NCHITTEST** contains the current mouse position in screen-relative coordinates (that is, relative to an origin at the top-left corner of the screen). The default window procedure responds to this message by testing this location against the various regions of the window and returning one of the result codes shown in Table 5–2. Most of the possible results refer to parts of the window's nonclient area, such as the title bar or sizing border. On receiving one of these results, Windows sends the window an appropriate nonclient-area mouse message, such as **WM_NCLBUTTONDOWN** or **WM_NCMOUSEMOVE**, with the hit-test result and the screen-relative mouse coordinates as parameters. These messages are really intended to be handled by the Windows system itself, and most programs simply pass them to the default window procedure instead of processing them explicitly.

Table 5–1. Mouse messages

Message type	Meaning
WM_NCHITTEST	Test mouse's location on screen
WM_LBUTTONDOWN	Left mouse button pressed in client area
WM_LBUTTONUP	Left mouse button released in client area
WM_LBUTTONDBLCLK	Left mouse button double-clicked in client area
WM_MBUTTONDOWN	Middle mouse button pressed in client area
WM_MBUTTONUP	Middle mouse button released in client area
WM_MBUTTONDBLCLK	Middle mouse button double-clicked in client area
WM_RBUTTONDOWN	Right mouse button pressed in client area
WM_RBUTTONUP	Right mouse button released in client area
WM_RBUTTONDBLCLK	Right mouse button double-clicked in client area
WM_NCLBUTTONDOWN	Left mouse button pressed in nonclient area
WM_NCLBUTTONUP	Left mouse button released in nonclient area
WM_NCLBUTTONDBLCLK	Left mouse button double-clicked in nonclient area
WM_NCMBUTTONDOWN	Middle mouse button pressed in nonclient area
WM_NCMBUTTONUP	Middle mouse button released in nonclient area
WM_NCMBUTTONDBLCLK	Middle mouse button double-clicked in nonclient area
WM_NCRBUTTONDOWN	Right mouse button pressed in nonclient area
WM_NCRBUTTONUP	Right mouse button released in nonclient area
WM_NCRBUTTONDBLCLK	Right mouse button double-clicked in nonclient area
WM_MOUSEMOVE	Mouse position changed within client area
WM_NCMOUSEMOVE	Mouse position changed within nonclient area

Table 5–2. Hit-test results

Name	Meaning
HTCAPTION	Title bar
HTREDUCE	Minimize box
HTZOOM	Maximize/restore box
HTMENU	Menu or menu bar
HTSYSMENU	System menu
HTBORDER	Nonsizing border
HTTOP	Sizing border, top edge
HTBOTTOM	Sizing border, bottom edge
HTLEFT	Sizing border, left edge
HTRIGHT	Sizing border, right edge
HTTOPLEFT	Sizing border, top-left corner
HTTOPRIGHT	Sizing border, top-right corner
HTBOTTOMLEFT	Sizing border, bottom-left corner
HTBOTTOMRIGHT	Sizing border, bottom-right corner
HTVSCROLL	Vertical scroll bar
HTHSCROLL	Horizontal scroll bar
HTSIZE	Size box
HTGROWBOX	Size box
HTCLIENT	Client area
HTTRANSPARENT	Window covered by another window
HTERROR	Screen background; beep
HTNOWHERE	Screen background; no beep

If the hit-test message returns a result of **HTCLIENT**, Windows generates a client-area mouse message, such as **WM_LBUTTONDOWN** or **WM_MOUSEMOVE**, instead of a nonclient-area message. In this case, before passing the mouse coordinates as a message parameter, it converts them to client-relative form, in which the origin of the coordinate system is the top-left corner of the window's client area instead of the full screen. The other parameter, **wParam**, is a flag word giving the state of all the mouse buttons and the keyboard modifier keys at the time of the click or move; it is thus analogous to the **modifiers** field of a Macintosh event record. The constants listed in Table 5–3 are bit masks for extracting the individual flags from this parameter.

Table 5–3. Mouse-key modifiers

Name	Meaning
MK_SHIFT	Shift key down
MK_CONTROL	Control key down
MK_LBUTTON	Left mouse button down
MK_MBUTTON	Middle mouse button down
MK_RBUTTON	Right mouse button down

Mouse Movements

On the Macintosh, mere movements of the mouse don't generate events in themselves. There is a mouse-moved event, but it's used strictly for cursor management: you get one only when the mouse moves outside a region that you've specified as valid for the current cursor shape. In Windows, you receive a mouse-moved message (**WM_MOUSEMOVE** or **WM_NCMOUSEMOVE**) every time the user moves the mouse. You can use this message to track the mouse when the user drags it while holding down one of the buttons.

Listing 5–1 shows an example in which the mouse is used to draw out a rectangle between two diagonally opposite corners within a window's client area. The tracking operation begins when we receive a **WM_LBUTTONDOWN** message reporting that the left mouse button was pressed inside the client area. We retrieve the window's client rectangle with the Windows function **GetClientRect**, call another Windows function, **ClipCursor**, to confine the mouse's movements within that rectangle, and save the current mouse coordinates as the starting point of the rectangle to be drawn. Then, each time we receive a **WM_MOUSEMOVE** message with the left button still down, we extend the rectangle to the new ending point. If a tracking operation is already in progress (indicated by the global program flag **tracking**), we first erase the rectangle drawn by the previous **WM_MOUSEMOVE** message; then we save the mouse coordinates as the new ending point, redraw the rectangle, and set the **tracking** flag to force the rectangle to be erased on the next iteration. When the user finally releases the button, we will receive a **WM_LBUTTONUP** message: we call **ClipCursor** again with a **NULL** parameter to cancel the confinement of the mouse, then clear the **tracking** flag to end the tracking operation. The next time the left button is pressed, a new tracking operation will begin.

Listing 5–**1.** Tracking the mouse

```
BOOL   tracking = FALSE;                                    // Currently tracking mouse?
POINT  startPoint;                                          // Starting point for tracking mouse
POINT  endPoint;                                            // Ending point for tracking mouse


LONG CALLBACK DoMessage (HWND thisWindow, UINT msgCode, WPARAM wParam, LPARAM lParam)

   // Get and process one message.

   {
      LONG  result = 0;                                     // Function result


      switch ( msgCode )                                    // Dispatch on message code
         {
            . . . ;

            case WM_LBUTTONDOWN:
               DoLButtonDown (thisWindow, wParam, lParam);  // Handle WM_LBUTTONDOWN message
               break;

            case WM_MOUSEMOVE:
               DoMouseMove (thisWindow, wParam, lParam);    // Handle WM_MOUSEMOVE message
               break;

            case WM_LBUTTONUP:
               DoLButtonUp (thisWindow, wParam, lParam);    // Handle WM_LBUTTONUP message
               break;

            . . . ;

            default:
               result = DefWindowProc (thisWindow, msgCode, // Pass message to Windows
                                       wParam, lParam);     //    for default processing
               break;

         }  /* end switch ( msgCode ) */


      return result;


   }  /* end DoMessage */
```

Listing 5–1. Tracking the mouse (*continued*)

```
VOID DoLButtonDown (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_LBUTTONDOWN message.

   {
      RECT  clientRect;                                    // Window's client rectangle


      GetClientRect (thisWindow, &clientRect);             // Get client rectangle in client coordinates
      /* Convert clientRect to screen-relative coordinates */

      ClipCursor (&clientRect);                            // Confine mouse movements to client rectangle

      startPoint.x = LOWORD(lParam);                       // Extract mouse coordinates
      startPoint.y = HIWORD(lParam);                       //    and save as starting point

   }  /* end DoLButtonDown */


VOID DoMouseMove (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_MOUSEMOVE message.

   {
      HDC  windowContext;                                  // Handle to device context for drawing in window


      if ( wParam & MK_LBUTTON );                          // Is left button down?
         {
            /* Get device context for drawing in window */;

            if ( tracking )                                // Have we already begun tracking?
               /* Erase previous rectangle from startPoint to endPoint */;

            endPoint.x = LOWORD(lParam);                   // Extract mouse coordinates
            endPoint.y = HIWORD(lParam);                   //    and save as ending point

            /* Draw new rectangle from startPoint to endPoint */;

            tracking = TRUE;                               // Indicate tracking in progress

            /* Release device context */;

         }  /* end if ( wParam & MK_LBUTTON ) */

   }  /* end DoMouseMove */
```

Listing 5–1. Tracking the mouse (*continued*)

```
VOID DoLButtonUp (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_LBUTTONUP message.

   {
      ClipCursor (NULL);                                    // Cancel mouse confinement

      tracking = FALSE;                                     // Indicate end of tracking sequence

   }  /* end DoLButtonUp */
```

Double Clicks

Windows considers a double mouse click to have occurred when the same mouse button is pressed twice in succession within a limited distance on the screen and a limited interval in time. The parameters defining these limits are set by the user with the Mouse control panel. You can find out the time interval with the Windows function **`GetDoubleClickTime`**, and the spatial limits by calling **`GetSystemMetrics`** with the selectors **`SM_CXDOUBLECLK`** and **`SM_CYDOUBLECLK`**. You can also change these values with the **`SystemParametersInfo`** function, using the selectors **`SPI_SETDOUBLECLKWIDTH`**, **`SPI_SETDOUBLECLKHEIGHT`**, and **`SPI_SETDOUBLECLKTIME`**, or with the function **`SetDoubleClickTime`**. However, as with the mouse speed and button swap, it's best not to disturb the user's control panel settings.

Double clicks will not be reported to a window unless you have specified the **`CS_DBLCLKS`** style option in the **`style`** field of the **`WNDCLASS`** parameter structure when registering the window's class. If you have, then the second click of a double-click sequence will be reported with a double-click message *in place of* the usual button-down message. Button-up messages are still reported normally. So, for instance, a double click of the left button will generate the following sequence of messages:

```
WM_LBUTTONDOWN
WM_LBUTTONUP
WM_LBUTTONDBLCLK
WM_LBUTTONUP
```

This method of reporting double clicks raises an interesting difficulty: there's no way to tell, when you receive the initial button-down message, whether it's the start of a double-click sequence or just an ordinary single click. Nor can you defer the decision until you receive the ensuing double-click message, since it may never arrive. The only reasonable way to handle the situation is to define the meaning of the first click as a self-contained operation and the second click as an extension of the same operation, in such a way that both can be processed independently when they arrive. In a text-editing program, for instance, you can use the first click to position the insertion caret between characters and the second to widen it into a full-word selection. If the second click never arrives, the first will already have been processed completely and correctly. (Luckily, WiniEdit doesn't have to deal with this issue, since all of its text selection is handled automatically by the system's built-in window procedure for edit controls.)

Capturing the Mouse

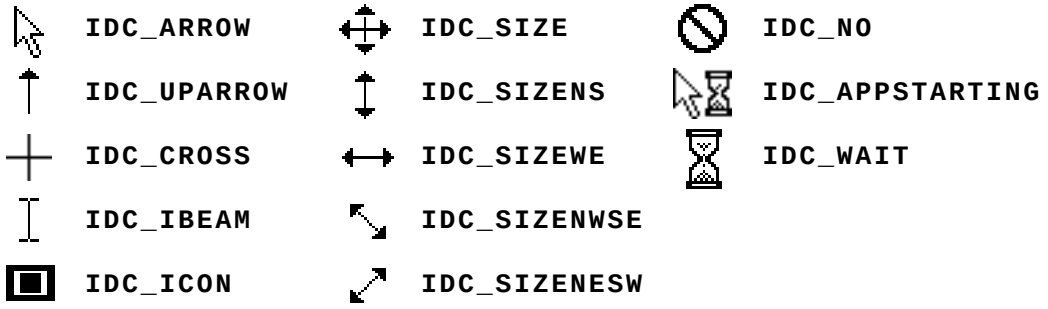
We mentioned earlier that there is an exception to the rule that mouse messages are always reported to the window in which they occur. In some circumstances, it may be appropriate for a window to *capture* the mouse, so that it will receive all mouse clicks and movements, even if they take place outside the window's borders. The usual reason for this is to continue tracking the mouse while the user holds down one of its buttons, even if it's dragged outside the window with the button still down. (Our example in Listing 5–1 addressed this issue in a different way, by confining the mouse within the window's client area with **`ClipCursor`**.) The Windows function **`SetCapture`** directs all mouse messages to a specified window until further notice; **`ReleaseCapture`** restores normal mouse reporting; **`GetCapture`** returns a handle to the window that has captured the mouse, if any.

Capturing the mouse is always a *temporary* state of affairs, to be used only for a limited time and a specific purpose. For instance, if you capture the mouse in order to track it during a drag, be sure to release it on receiving the mouse-up message ending the tracking operation: otherwise, no other window will be able to receive mouse input and the user will be unable to activate another window by clicking in it. A thread can capture the mouse only for one of its own windows, never for a window belonging to another thread. The capturing window should always be the active (frontmost) window on the screen. If it isn't, the capture will not behave as expected: mouse events will be clipped to the window's visible region, with those outside the region going unreported.

The Cursor

Windows cursors are pretty much the same as their Macintosh counterparts: they have a color or monochrome bit image, a monochrome bit mask, and a hot spot. The image defines the cursor's appearance on the screen, the mask works like a cookie-cutter to define its shape, and the hot spot defines the point within the image that marks the actual mouse location. However, because Windows has to work with a variety of display devices at different resolutions and aspect ratios, the dimensions of the cursor aren't standardized, as they are on the Macintosh at 16 pixels by 16. The cursor's width and height aren't even necessarily equal, since some devices use square pixels and others use rectangular ones with different horizontal and vertical resolutions. The cursor dimensions for the currently configured display device are available via the **`GetSystemMetrics`** function, using the selectors **`SM_CXCURSOR`** and **`SM_CYCURSOR`**.

Figure 5–1. Stock cursors



You can create a cursor dynamically at run time with the Windows function **CreateCursor**, but the usual practice is to read it in as a resource with **LoadCursor**. Both functions return a cursor handle of type **HCURSOR**. Like all resource-loading functions, **LoadCursor** takes two parameters: a handle to the program instance in whose executable file the resource resides, and a second parameter identifying the individual resource within the file. As we learned in Chapter 2, this second parameter is nominally a string pointer to the resource name, but in reality it can instead be an integer resource ID converted to string form with the **MAKEINTRESOURCE** macro. A null instance handle refers to one of the standard or "stock" cursors shown in Figure 5–1; the second parameter must then be one of the constants in Table 5–4. (The prefix **IDC_** stands for "identifier, cursor.") These constants are already typecast into string pointers, so it isn't necessary to use **MAKEINTRESOURCE** on them. For example, the statement

```
arrowCursor = LoadCursor (NULL, IDC_ARROW);
```

loads the standard arrow cursor. The stock cursors actually exist in a variety of dimensions and color depths for use on different display devices; **LoadCursor** automatically selects the one best suited to the current screen.

Each time the user moves the mouse within a window's client area, Windows sends the window the message **WM_SETCURSOR**, with the hit-test code returned by **WM_NCHITTEST** as one of its parameters. By processing this message, you can test the cursor's location and set its shape accordingly with the Windows function **SetCursor**. The standard window procedure does this automatically for the window's non-client area, depending on the hit-test code: for instance, if the hit-test value is **HTTOPLEFT**, denoting the top-left corner area of the window's sizing border, Windows will set the cursor to **IDC_SIZENWSE**, the double-headed arrow pointing northwest and southeast.

The easiest way to manage the cursor, though, is to specify a *class cursor* when you register your window class. Windows will automatically set the cursor to this shape whenever it moves into the client area of a window belonging to the class. If you need to divide your window into distinct functional areas with different cursor shapes, you can do it by creating a "transparent" child window (one with no visible border) for each area, and giving each such child window a different class cursor. When the child window receives the **WM_SETCURSOR** message, it will relay it back to the parent window. If it wishes, the parent window itself can set the cursor and return a result of **TRUE** to indicate that the cursor is already taken care of; or it can return **FALSE** and let the child window do the job.

Table 5–4. Stock cursors

Name	Description	Purpose
IDC_ARROW	Northwest arrow	General-purpose
IDC_UPARROW	Up arrow	General-purpose
IDC_CROSS	Crosshairs	Precise location
IDC_IBEAM	I-beam	Text selection
IDC_ICON	Empty icon	Program selection
IDC_SIZE	Four-way arrow	Move or size window
IDC_SIZEALL	Four-way arrow	Move or size window
IDC_SIZENS	North–south double arrow	Size window, top or bottom edge
IDC_SIZEWE	West–east double arrow	Size window, left or right edge
IDC_SIZENWSE	Northwest–southeast double arrow	Size window, top-left or bottom-right corner
IDC_SIZENESW	Northeast–southwest double arrow	Size window, top-right or bottom-left corner
IDC_NO	Circle with diagonal slash	Prohibited operation
IDC_APPSTARTING	Arrow with hourglass	Waiting for program to load
IDC_WAIT	Hourglass	General delay

> If you decide to do your cursor management by hand, be sure to set the class cursor to **NULL** when you register your window class; otherwise, the cursor will flicker on the screen as your program and Windows fight a tug-of-war over its shape every time the user moves the mouse. Also, remember that the cursor is a system-wide resource that you share with other programs. Don't change its shape unless it's within the client area of one of your windows or unless you've captured the mouse; and if you do capture the mouse, remember to restore the cursor to its previous shape before releasing it.

You can get a handle to the current cursor with the Windows function **GetCursor** and retrieve or change its coordinates on the screen with **GetCursorPos** or **SetCursorPos**. On a system without a mouse, **SetCursorPos** is handy for moving the cursor around on the screen with the keyboard arrow keys. The **ClipCursor** function confines the cursor within a specified rectangle, pinning it at the rectangle's edges and preventing it from traveling outside. (We've already seen an example of this function in action in Listing 5–1.) You can find out the current confinement rectangle, if any, with **GetClipCursor**; this is useful for saving and restoring the previous confinement state.

The Windows function **ShowCursor** hides or shows the cursor, depending on the value of a boolean parameter. Like its Macintosh counterpart, the Windows cursor maintains an integer visibility level that records how many times it has been hidden and not yet reshown. Hiding the cursor decrements the visibility level by one; showing it increments the level. A negative visibility level makes the cursor invisible on the screen. The visibility level is ordinarily initialized to **0**, but on systems without a mouse, it's initialized to **-1** instead, making the cursor invisible by default.

If you've created a cursor from scratch with **`CreateCursor`**, you must destroy it with **`DestroyCursor`** before exiting from your program. It isn't necessary to destroy a cursor that you've loaded as a resource with **`LoadCursor`**.

The Keyboard

The other primary input device on all Windows systems is the keyboard. The keyboard is designed mainly for entering text, but it can also be used in place of the mouse for manipulating menus, windows, and controls on the screen. Although it's possible, as WiniEdit demonstrates, to write a program that never interacts explicitly with the keyboard, most real-world programs will need to receive and process keyboard input in some form. Table 5–5 lists the main message types related to keyboard input; we'll be discussing them in the remainder of this section.

Table 5–5. Keyboard messages

Message type	Meaning
`WM_KEYDOWN`	Key pressed
`WM_KEYUP`	Key released
`WM_SYSKEYDOWN`	Key pressed with Alt key
`WM_SYSKEYUP`	Key released with Alt key
`WM_CHAR`	Character typed
`WM_DEADCHAR`	Dead character typed
`WM_SYSCHAR`	Character typed with Alt key
`WM_SYSDEADCHAR`	Dead character typed with Alt key
`WM_SETFOCUS`	Window acquiring input focus
`WM_KILLFOCUS`	Window losing input focus

Input Focus

No more than one window at a time can be the target of the user's keyboard input. The window so designated is said to have the *input focus.* The window with the focus is always either the active window or one of its children. In a dialog box, for instance, the focus normally belongs to one of the dialog's controls, identified by a dotted *focus rectangle* around the text of a button control or an insertion caret or selection highlight in an edit control. The standard window procedure responds to the keystrokes Tab and Shift-Tab by cycling the focus forward and backward among the dialog's controls, and to the space bar by activating or toggling the button with the focus as if it had been clicked with the mouse.

A window normally acquires the input focus when it becomes the active window. You can also set the focus explicitly with the Windows function **`SetFocus`**, or find out which window has it with **`GetFocus`**. Windows signals when the focus shifts from one window to another by sending the message **`WM_KILLFOCUS`** to the window losing the focus and **`WM_SETFOCUS`** to the one gaining it. WiniEdit's document window, for instance, responds to this message by relaying the focus to its child, the edit control (see Listing 5–2). The edit control's built-in window procedure

Listing 5–2. Handle `WM_SETFOCUS` message

```
VOID DoSetFocus (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_SETFOCUS message.

   {
      SetFocus (TheEditor);                        // Pass focus to the edit control

   }  /* end DoSetFocus */
```

will then receive and handle all keyboard input for as long as the document window remains active.

A window in the minimized state (reduced to a button in the Windows 95 task bar) can't respond to keyboard input. In this case, no window is considered to have the input focus: all keyboard input is then directed to the system itself, rather than to any individual window.

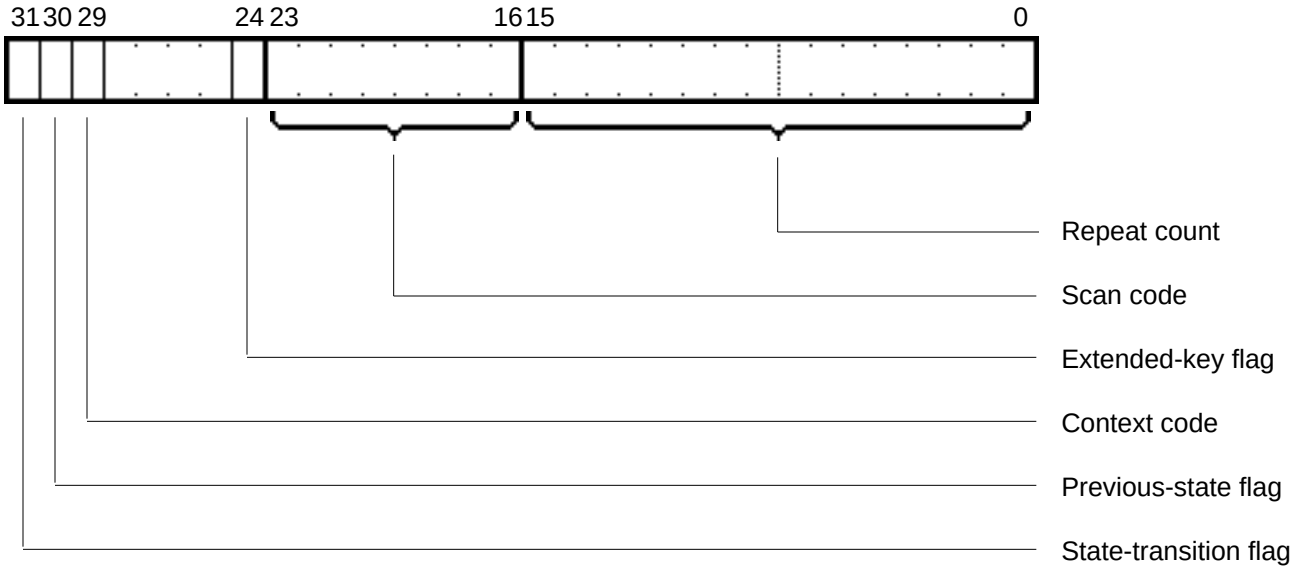
Keystroke Messages

Every key on the keyboard is identified by a device-dependent *scan code* that identifies the key to the keyboard driver. The keyboard driver converts this scan code into a device-independent *virtual-key code*, using a keyboard layout configured by the user with the Keyboard control panel. The virtual-key code identifies the key by its purpose instead of its physical location on the keyboard, such as **VK_A** for the letter **A**, **VK_ENTER** for the Enter key, or **VK_SPACE** for the space bar. (See the *Win32 Programmer's Reference* for a complete table of virtual-key codes.)

When the user presses or releases a key, the keyboard driver posts a message, normally a **WM_KEYDOWN** or **WM_KEYUP**, to the global system message queue. (If the Alt key was being held down at the time, or if no window currently has the input focus, the message is **WM_SYSKEYDOWN** or **WM_SYSKEYUP** instead.) The message identifies the key by both its physical scan code and the corresponding virtual-key code, along with some other information that we'll talk about in a minute. The Windows system then moves the message from the system message queue into the private queue of the currently active thread. Eventually, the thread's message loop will retrieve and process the message.

All of these *keystroke messages* (**WM_KEYDOWN**, **WM_KEYUP**, **WM_SYSKEYDOWN**, **WM_SYSKEYUP**) receive the virtual-key code as their **wParam** parameter; the **lParam** parameter contains a variety of information packed into a 32-bit long word (see Figure 5–2):

Figure 5–2. **`lparam`** parameter for keystroke messages



- The 8-bit raw scan code generated by the keyboard hardware.
- A 16-bit *repeat count.* Before posting a keyboard message to the queue, Windows looks first to see if there's one already there for the same keystroke. If so, it just increments the existing message's repeat count instead of posting another separate message. This typically happens when the user presses and holds a key, generating auto-repeat keystrokes faster than the program can retrieve them from the queue.
- A 1-bit *previous-state flag* showing whether the key was previously up or down. If this flag is set, it shows that the current message is part of a continuing series generated by an auto-repeating key.
- A 1-bit *state-transition flag* that tells whether the message was generated by the user's pressing a key (**`WM_KEYDOWN`** or **`WM_SYSKEYDOWN`**) or releasing one (**`WM_KEYUP`** or **`WM_SYSKEYUP`**).
- A 1-bit *extended-key flag* indicating whether the keystroke was generated by one of the additional keys on the extended keyboard. Table 5–6 lists the extended keys that cause this flag to be set. In general, they are those in the extra clusters to the right of the main keyboard layout, such as the arrow keys and the keypad. For some reason, however, not all of the keypad keys are considered extended keys: only the Enter, Divide (**`/`**), and Num Lock keys are included. Most programs don't pay much attention to the extended-key flag, but it can sometimes be useful for distinguishing between otherwise functionally equivalent keys, such as the left and right Ctrl or Alt keys or the keyboard and keypad Enter keys.
- A 1-bit *context code* that tells whether the Alt key was down at the time of the keystroke. In the case of a system keystroke (**`WM_SYSKEYDOWN`** or **`WM_SYSKEYUP`**), this flag distinguishes whether the keystroke is directed to the system because of the Alt key or because no window has the input focus.

Table 5–6. Extended keys

Virtual-key code	Key
VK_MENU	Right-hand Alt key
VK_CONTROL	Right-hand Ctrl key
VK_INSERT	Insert
VK_DELETE	Delete
VK_HOME	Home
VK_END	End
VK_PRIOR	Page Up
VK_NEXT	Page Down
VK_LEFT	Left arrow
VK_RIGHT	Right arrow
VK_UP	Up arrow
VK_DOWN	Down arrow
VK_RETURN	Keypad Enter key
VK_DIVIDE	Keypad Divide (**/**)
VK_NUMLOCK	Num Lock
VK_SNAPSHOT	Print Screen (**F13**)
VK_SCROLL	Scroll Lock (**F14**)
VK_PAUSE	Pause/Break (**F15**)

In general, raw keystroke messages (as opposed to the character-oriented messaages discussed in the next section) are most useful for handling non-character input such as the arrow and function keys. If you do need to process raw keystrokes, you should normally confine yourself to the non-system messages, **WM_KEYDOWN** and **WM_KEYUP**. The default window procedure, **DefWindowProc**, ignores these messages, but it uses the system versions, **WM_SYSKEYDOWN** and **WM_SYSKEYUP**, to implement many of the standard keyboard conventions that Windows users expect, such as the Alt-key interface for menu selection that we'll be discussing in Chapter 9. If the default window procedure doesn't receive system keystrokes, your user will lose access to these standard Windows features. So if you absolutely must intercept system keystrokes yourself, be sure you pass them on to **DefWindowProc** *in addition to* any processing you may be doing on your own. Your users will thank you.

Character Messages

On the Macintosh, a single keyboard event (key-down, key-up, or auto-key) contains codes identifying both the physical key that was pressed and the logical character it represents. In Windows, the low-level keystroke messages that we've discussed so far contain no reference to specific text characters. While it's true that many virtual-key codes refer to character keys, such as **VK_M** or **VK_8** or **VK_SPACE**, that doesn't mean that every keystroke generated by such a key

necessarily represents that particular character. The **VK_M** key, for instance, may produce an uppercase **M** or a lowercase **m**, depending on the state of the Shift and Caps Lock keys; **VK_8** can produce either a numeral **8** or an asterisk (*****); or a keystroke may have been typed with the Alt or Ctrl key down and not represent any character at all. The task of mapping raw keystrokes into characters is performed by the Windows function **TranslateMessage**.

We've already encountered **TranslateMessage** in Chapter 3 as part of our NullApp program's main message loop routine (reproduced for reference in Listing 5–3). After retrieving a message with **GetMessage**, we pass it to **TranslateMessage** to see if it's a keystroke representing a text character. If so, **TranslateMessage** will generate a corresponding *character message* (**WM_CHAR** or **WM_SYSCHAR**) and post it to the message queue. The character message goes straight to the front of the queue instead of the end, so it doesn't get out of order with other possible keyboard messages that may be pending in the queue. **TranslateMessage** simply ignores any message that isn't a keystroke.

All character messages identify the character typed via a character code in the message's **wParam** parameter. The **lParam** parameter holds the same information as in a raw keystroke message, as shown earlier in Figure 5–2. The contents of **lParam** are simply copied directly from the keystroke message (**WM_KEYDOWN** or **WM_SYSKEYDOWN**) that gave rise to the character message. Most of this information is normally of no interest, but don't forget to check the repeat count in case several strokes of the same key have been combined into a single message.

Polling the Keyboard

In determining what character a keystroke represents, **TranslateMessage** must take into account the state of the Shift and Caps Lock keys. If you look carefully, however, you'll find that this information is not included anywhere in the keyboard message's parameters: there is no equivalent to the **modifiers** field of a Macintosh event record. Instead, **TranslateMessage** has to poll the state of the modifier keys directly, using the Windows function **GetKeyState**. This function accepts a virtual-key code as a parameter and reports whether the key is up or down. For keys that toggle a persistent state, such as Caps Lock, Num Lock, and Scroll Lock, it also reports whether the state is currently toggled on or off. Note that this function applies to just a single designated key; a related function, **GetKeyboardState**,

Listing 5–3. NullApp main program loop

```
VOID MainLoop (VOID)

   // Execute one pass of main program loop.

   {
      MSG   theMessage;                              // Next message to process


      ContinueFlag = GetMessage(&theMessage, NULL, 0, 0);   // Get next message

      TranslateMessage (&theMessage);                // Convert virtual keys to characters
      DispatchMessage  (&theMessage);                // Send message to window procedure

   }  /* end MainLoop */
```

returns an array of state information for the entire keyboard.

You might think that polling the keyboard this way is an unreliable way of checking the modifier keys for a keyboard event, since the state of the keys may have changed between the time the event was posted and the time it's processed. It turns out, however, that the **`GetKeyState`** and **`GetKeyboardState`** functions keep track of such changes and report the state of the keyboard *at the time the current keyboard message was posted*, not its current state at the time it's actually polled. This makes these functions ideal for processing keystrokes one by one out of the queue, but it also means you can't rely on them for polling the *current* state of the keyboard, right this instant. Luckily, there's yet another Windows function, **`GetAsyncKeyState`**, that reports the state of a key *right now*, without reference to any previously posted message.

Dead Keys

In some keyboard layouts, some of the keys may generate accents or diacritical marks for use in languages other than English, such as an acute accent (´), umlaut (¨), or circumflex (ˆ). Such characters are intended to combine with the next character typed to produce an accented character, such as **é**, **ü**, or **ô**. The accented letter is represented by a character code of its own, different from those of both the accent itself and the base letter to which it attaches. (That is, the character code for **é** is different from those for ´ and **e**.) Since the accent key by itself doesn't generate a text character until after the next keystroke, it is known as a *dead key;* the accent character it represents, which combines with the next character typed to form a single character, is called a *dead character*.

When the Windows **`TranslateMessage`** function determines that a keystroke it has received represents a dead character, it generates a message of type **`WM_DEADCHAR`** (or **`WM_SYSDEADCHAR`** for a system keystroke) instead of the usual **`WM_CHAR`** message. Then, when it receives the next keystroke, it combines that character with the dead character and generates a **`WM_CHAR`** message carrying the character code for the resulting combination character (**é** in the previous example). Thus the overall sequence of messages generated goes like this:

```
WM_KEYDOWN                                          WM_SYSKEYDOWN
WM_DEADCHAR (´)                                     WM_SYSDEADCHAR (´)
WM_KEYUP        or, for a system keystroke,         WM_SYSKEYUP
WM_KEYDOWN                                          WM_SYSKEYDOWN
WM_CHAR (é)                                         WM_SYSCHAR (é)
WM_KEYUP                                            WM_SYSKEYUP
```

If the next character typed after the dead character is not one that it can combine with (such as a consonant instead of a vowel following an acute accent), **`TranslateMessage`** treats them as separate characters and generates *two* **`WM_CHAR`** messages. For example, typing an acute accent followed by a lowercase **m** would result in the following sequence of messages:

```
WM_KEYDOWN                                          WM_SYSKEYDOWN
WM_DEADCHAR (´)                                     WM_SYSDEADCHAR (´)
WM_KEYUP                                            WM_SYSKEYUP
WM_KEYDOWN                    or                    WM_SYSKEYDOWN
WM_CHAR (´)                                         WM_SYSCHAR (´)
WM_CHAR (m)                                         WM_SYSCHAR (m)
WM_KEYUP                                            WM_SYSKEYUP
```

Unless you have some special reason for caring about them, it's usually safe to ignore dead-character messages: the dead character will always turn up eventually in an ordinary character message, either in its own right or as part of a combination character like **é**.

Mouse and Keyboard Operations

Table 5–7 summarizes some common Windows functions relating to mouse and keyboard input. We've already discussed some of them in this chapter; you can learn about the rest in the *Win32 Programmer's Reference.*

Table 5–7. Common mouse and keyboard functions

Function	Mac counterpart	Purpose
`SetCapture`	————	Capture mouse
`GetCapture`	————	Get window that has captured the mouse
`ReleaseCapture`	————	Release captured mouse
`GetDoubleClickTime`	`GetDblTime`	Get double-click interval
`SetDoubleClickTime`	————	Set double-click interval
`SwapMouseButton`	————	Reverse left and right mouse buttons
`CreateCursor`	————	Create new cursor
`LoadCursor`	`GetCursor`	Load cursor from template resource
`CopyCursor`	————	Make copy of cursor
`DestroyCursor`	————	Destroy cursor
`GetCursor`	————	Get current cursor
`SetCursor`	`SetCursor`	Set current cursor
`SetSystemCursor`	————	Change appearance of stock cursor
`GetCursorPos`	`GetMouse`	Get current cursor coordinates
`SetCursorPos`	————	Set cursor coordinates
`ShowCursor`	`ShowCursor`, `HideCursor`	Make cursor visible or invisible
`ClipCursor`	————	Confine cursor within a rectangle
`GetClipCursor`	————	Get current confinement rectangle
`GetKeyState`	————	Get state of key at time of message
`GetKeyboardState`	`GetKeys`	Get state of full keyboard at time of message
`GetAsyncKeyState`	————	Get current state of key
`GetKeyNameText`	————	Get name of key in string form
`GetFocus`	————	Get window with input focus
`SetFocus`	————	Set window with input focus

The Same Thing...

Mouse

- A Macintosh program receives a mouse-down or mouse-up event when the user presses or releases the mouse button.
- A Windows program receives a button-down or button-up message when the user presses or releases a mouse button.

Cursor

- The Macintosh cursor has an image, a mask, and a hot spot.
- A Macintosh cursor can be created dynamically or read in as a resource.
- The Macintosh Toolbox has certain standard cursors built in as system resources and available to all programs.
- The Macintosh cursor maintains an integer visibility level to count how many times the cursor has been hidden and not yet reshown.
- The Windows cursor has an image, a mask, and a hot spot.
- A Windows cursor can be created dynamically or read in as a resource.
- Windows has certain standard (“stock”) cursors built in as system resources and available to all programs.
- The Windows cursor maintains an integer visibility level to count how many times the cursor has been hidden and not yet reshown.

Keyboard

- A Macintosh program receives a key-down or key-up event when the user presses or releases a key on the keyboard.
- A Windows program receives a key-down or key-up event when the user presses or releases a key on the keyboard.

...Only Different

Mouse

- The mouse (or other pointing device) is an integral part of every Macintosh system.
- The Macintosh mouse always has exactly one button.
- The Windows mouse is an optional accessory; not every system has one.
- The Windows mouse can have one, two, or three buttons.

• Macintosh mouse events are directed globally to the program as a whole.	• Windows mouse messages are directed specifically to the window in which they occur.
• Because mouse clicks are global to the entire program, the Macintosh has no concept of capturing the mouse.	• In Windows, a window can capture the mouse, receiving all subsequent mouse messages (even those occurring outside the given window) until it releases the mouse.
• Macintosh programs receive the same type of mouse-down event for any mouse click, no matter where it occurs on the screen.	• Windows programs receive different types of button-down messages, depending on whether the mouse was clicked in a window's client or nonclient area.
• Macintosh programs must respond explicitly to all mouse clicks, including those in a window's frame (title bar, size box, close box, zoom box) or in the menu bar.	• Windows programs need only respond to mouse clicks in a window's client area; those in the nonclient area (title bar, sizing border, menu bar, close box, minimize and maximize boxes) are handled automatically by the default window procedure.
• On the Macintosh, mere movements of the mouse don't generate events in themselves.	• Windows programs receive a mouse-moved message whenever the user moves the mouse.
• Macintosh programs must poll the mouse position in order to track its movements.	• Windows programs can use mouse-moved messages to track the mouse's movements.
• Macintosh programs must decide for themselves when a double click has occurred, by comparing the times and locations of consecutive mouse clicks.	• Windows programs receive a double-click message to tell them when a double click has occurred.

Cursor

• The Macintosh cursor is always the same size, 16 pixels wide by 16 high.	• The Windows cursor can vary in dimensions according to the characteristics (resolution and aspect ratio) of the screen on which it is displayed.

- A Macintosh program must manage the appearance of the cursor in all parts of the screen.
- A Windows program must manage the appearance of the cursor only in a window's client area; the Windows system manages it automatically in the nonclient area or outside the window.
- Macintosh programs manage the cursor either by polling its location on every pass of the event loop or by passing a valid region for the current cursor as a parameter to **`WaitNextEvent`**.
- Windows programs manage the cursor by receiving a **`WM_SETCURSOR`** message every time the user moves the mouse, or by specifying a default cursor for an entire class of windows.

Keyboard

- The Macintosh Toolbox reports both a physical key code and a logical character code in a single event.
- Windows sends two different types of keyboard messages: keystroke messages containing a physical key code and character messages containing a logical character code.
- Macintosh keyboard events are directed globally to the program as a whole.
- Windows keyboard messages are directed specifically to the window that has the input focus.
- Every Macintosh keyboard event represents a single keystroke, whether key-down, key-up, or auto-key.
- A Windows keyboard message has a repeat count, allowing it to represent several successive strokes of the same key.

Chapter 6

The Graphics Device Interface

Just as the QuickDraw graphics routines form one of the cornerstones of the Macintosh Toolbox, the Windows user interface rests on the foundation of the Graphics Device Interface (GDI). Everything a Windows user sees on the screen is ultimately put there through the agency of a GDI routine. All of the familiar features of QuickDraw—points and rectangles, bitmaps and graphics ports, pens and patterns, lines and shapes—have their counterparts in the Windows GDI. In this chapter, we'll present an overview of the important concepts underlying GDI. It's a big subject, though, and a complete discussion would be beyond the scope of this book. As always, your best source for all the details and intricacies is the *Win32 Programmer's Reference*.

Graphical Data Structures

Any graphics system, whether it's QuickDraw, GDI, or anything else, has to deal with the same basic data objects: points on the coordinate plane, rectangular boxes, more elaborately shaped regions, and color values. It's no surprise, then, that the Windows versions of these objects closely resemble their Macintosh counterparts. Before we delve into GDI's graphical capabilities, let's take a quick look at how Windows defines these fundamental graphical objects and the basic operations you can perform on them.

Points

What is there to say about a point? It has a horizontal and a vertical coordinate, both of which are integers. Unlike QuickDraw, Windows names and orders its coordinates according to accepted mathematical conventions: **x** and **y**, rather than **v** and **h**. The coordinates are independent of any particular origin or scale: as we'll see later, these can vary from one coordinate system to another.

In the Win32 version of the generic **POINT** structure (Listing 6–1), the coordinates are long (32-bit) integers; earlier versions of the system used short (16-bit) integers. There's also a pair of related structures, **POINTL** and **POINTS**, that define the coordinates specifically as long and short integers, respectively, as well as derived data types like **PPOINT** (pointer to point) and **LPPOINT** (long pointer to point). The system macros **POINTTOPOINTS** and **POINTSTOPOINT** convert a point between long and short coordinates. Another macro,

Listing 6–1. Point structure

```
typedef struct tagPOINT
   {
      LONG  x;                          // Horizontal coordinate
      LONG  y;                          // Vertical coordinate

   }  POINT;
```

MAKEPOINTS, converts a long integer to a point, interpreting the low-order word as the **x** coordinate and the high-order word as the **y**.

Rectangles

Four coordinates define a rectangle by its left, top, right, and bottom edges (Listing 6–2). The coordinates must be specified in the correct order: that is, the left edge must be less than the right and the top must be less than the bottom. Table 6–1 lists some common utility operations that are available for working with rectangles:

- **SetRect** accepts a pointer to a rectangle structure and sets its coordinates to four specified values.
- **CopyRect** copies the coordinates from one rectangle to another.
- **SetRectEmpty** sets all four of a rectangle's coordinates to zero.
- **IsRectEmpty** tests whether a rectangle is empty: that is, whether its left edge is beyond its right (**left** ≥ **right**) or its top edge is below its bottom (**top** ≥ **bottom**).
- **PtInRect** tests whether a given point lies within a given rectangle.
- **EqualRect** tests whether two rectangles have identical coordinates.
- **InflateRect** expands or contracts a rectangle by moving opposite edges (left and right or top and bottom) inward or outward by the same amount with respect to the center. It can adjust either the width of the rectangle, its height, or both independently.
- **OffsetRect** moves a rectangle to a new position on the coordinate plane without altering its dimensions.
- **UnionRect** forms the union of two rectangles: the smallest rectangle that completely encloses both of them.
- **IntersectRect** forms the intersection of two rectangles: the largest rectangle completely enclosed within both of them.
- **SubtractRect** forms the difference of two rectangles: the largest rectangle completely enclosed within the first that doesn't intersect the second.

Listing 6–2. Rectangle structure

```
typedef struct tagRECT
   {
      LONG  left;                          // Left edge
      LONG  top;                           // Top edge
      LONG  right;                         // Right edge
      LONG  bottom;                        // Bottom edge

   }  RECT;
```

Table 6–1. Common rectangle functions

Function	Mac counterpart	Purpose
`SetRect`	`SetRect`	Set rectangle coordinates
`CopyRect`	————	Copy coordinates from one rectangle to another
`SetRectEmpty`	————	Set rectangle to empty (all coordinates zero)
`IsRectEmpty`	`EmptyRect`	Is rectangle empty?
`PtInRect`	`PtInRect`	Is given point in given rectangle?
`EqualRect`	`EqualRect`	Are two rectangles identical?
`InflateRect`	`InsetRect`	Expand or contract size of rectangle
`OffsetRect`	`OffsetRect`	Change position of rectangle
`UnionRect`	`UnionRect`	Form union of two rectangles
`IntersectRect`	`SectRect`	Form intersection of two rectangles
`SubtractRect`	————	Form difference of two rectangles

Regions and Paths

Windows has objects called *regions,* but they aren't quite as general as the QuickDraw regions you're familiar with from the Macintosh. Windows regions are limited to the following standard shapes, either singly or in combination:

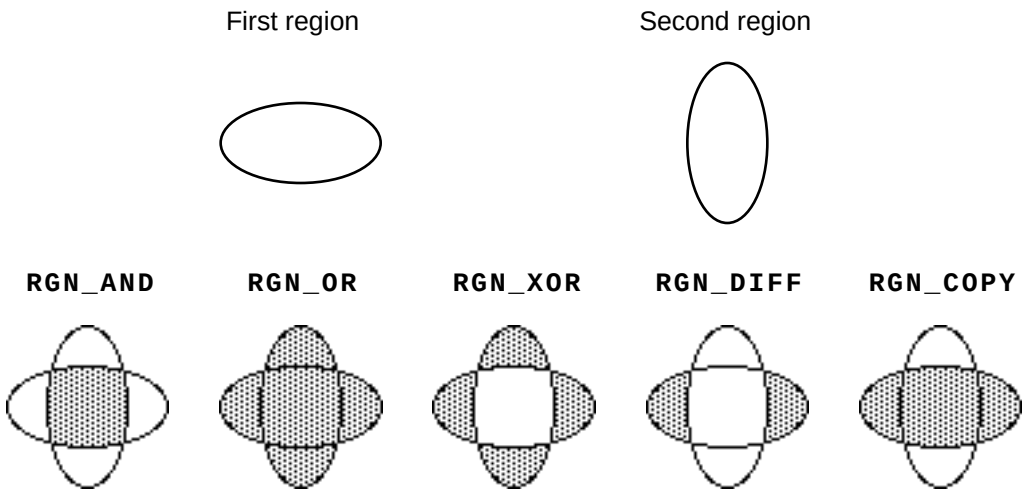
- Rectangles
- Rounded-corner rectangles
- Ellipses (called ovals on the Macintosh)
- Polygons

Each of the functions listed in Table 6–2 creates a *simple region* in one of these shapes and returns a *region handle* of type **`HRGN`**. You can use the resulting region as is, or you can combine it with other regions to build a *complex region.* The **`CombineRgn`** function accepts a pair of source regions, along with one of the *combining modes* listed in Table 6–3, and sets a designated destination region to the resulting combination. Figure 6–1 illustrates the results. **`CombineRgn`** also returns a coded integer telling whether the result of the combining operation is an empty

Table 6–2. Region creation functions

Function	Purpose
`CreateRectRgn`	Create rectangular region with given coordinates
`CreateRectRgnIndirect`	Create rectangular region equivalent to a given rectangle
`CreateRoundRectRgn`	Create rounded-rectangular region with given coordinates and corner diameters
`CreateEllipticRgn`	Create elliptical region with given coordinates
`CreateEllipticRgnIndirect`	Create elliptical region inscribed in given rectangle
`CreatePolygonRgn`	Create polygonal region with given vertices

Figure 6–1. Combining regions



region, a simple region, a complex region, or an error.

You can apply any of the standard drawing operations to a region, as we'll see later in the section on shape drawing. We'll also see how you can use a region as a clipping boundary to confine the results of your other drawing operations. You can move a region to a new location on the coordinate plane with the Windows function **`OffsetRgn`**, retrieve its bounding box with **`GetRgnBox`**, compare two regions with **`EqualRgn`**, and test whether a region includes a given point with **`PtInRegion`**.

The true Windows equivalent of a general Macintosh region is a *path.* Unlike a Windows region, a path isn't built up from a limited selection of fixed shapes; it can be any shape you can construct with GDI drawing operations. You define a path in essentially the same way as a Macintosh region:

1. Open a new path definition by calling the Windows function **`BeginPath`** (analogous to the Macintosh QuickDraw routine **`OpenRgn`**).
2. Draw the outline of the path, using any sequence of drawing operations.
3. Conclude the path definition by calling **`EndPath`** (like **`CloseRgn`** on the Macintosh).

Once you've defined a path, you can *stroke* it (draw its outline with a graphics pen), *fill* its interior with a pattern (provided it's a closed path), or both. You can convert a path to an equivalent region with the Windows function **`PathToRegion`**, or to a series of line segments

Table 6–3. Region combining modes

Mode	Meaning
`RGN_AND`	Intersection of two regions
`RGN_OR`	Union of two regions
`RGN_XOR`	Union minus intersection of two regions
`RGN_DIFF`	Difference of two regions
`RGN_COPY`	Copy of first region; second ignored

with **FlattenPath**. We'll see later how to use a path as a clipping boundary, either by itself or in combination with an existing region.

Colors

The basic Windows data type for representing color values is **COLORREF**, which is simply a 32-bit long word containing three byte-length RGB color components in the format

```
00bbggrr
```

where **rr** is the red component, **gg** the green, and **bb** the blue. Since each of the three components is a single byte, they can range over 256 possible values, from **0x00** to **0xFF**. The high-order byte is unused, and must always be set to **0x00**. A **COLORREF** value of **0x00000000** represents pure black, **0x00FFFFFF** pure white.

The Windows interface defines a utility macro named **RGB**, which accepts three byte-length integers representing the color components and packs them together into a **COLORREF**:

```
rgbColor = RGB(redValue, greenValue, blueValue);
```

There's also a set of complementary macros for extracting the individual components of a **COLORREF**:

```
redValue   = GetRValue(rgbColor);
greenValue = GetGValue(rgbColor);
blueValue  = GetBValue(rgbColor);
```

Device Contexts

The Windows equivalent of a QuickDraw graphics port is a *device context*. Like a graphics port, a device context is a complete graphical environment, containing all the information GDI needs to manage its drawing operations: a pen for line drawing, brushes (like Macintosh patterns) and colors for area fill, font information for drawing text, a clipping region, and so forth. All drawing operations must take place within a specified device context.

As on the Macintosh, the most common type of device context that most programs use is associated with a window on the screen. Other types are used for drawing to the full screen (rather than just a window), to a printer or other hardcopy device, or to an offscreen bitmap in memory. We'll be confining our discussion here primarily to drawing in a window, but most of what we say carries over to the other types of device context as well.

One important difference between QuickDraw and GDI is that GDI has no concept of an implicit "current device context," corresponding to the QuickDraw current port. Instead, all GDI functions accept an explicit *device context handle* (type **HDC**) as one of their parameters. We'll learn in a minute how to obtain a device context handle; once you have one, you must supply it as a parameter whenever you want to perform any drawing operation within that context.

Device Context Attributes

Like a QuickDraw graphics port, a device context includes a great many properties and attributes that control various aspects of its drawing behavior. We'll be discussing most of these attributes in the remainder of this chapter; they include

- A bitmap for drawing in
- A mapping mode that defines the logical coordinate space in which drawing operations take place
- A geometric transformation matrix for mapping logical drawing coordinates to those of the physical device
- A clipping region that confines the results of all drawing operations
- A pen for drawing lines
- A pen position in logical coordinates
- Pen and background modes governing how the pixels of the pen combine with the existing pixels of the bitmap

Table 6–4. Device context access functions

Attribute	Retrieval function	Change function
Destination bitmap	`GetCurrentObject`	`SelectObject`
Mapping mode	`GetMapMode`	`SetMapMode`
Transformation matrix	`GetWorldTransform`	`SetWorldTransform`
Clipping region	`GetClipRgn`	`SelectClipRgn`, `SelectClipPath`
Pen	`GetCurrentObject`	`SelectObject`
Pen position	`GetCurrentPositionEx`	`MoveToEx`
Pen mode	`GetROP2`	`SetROP2`
Background mode	`GetBkMode`	`SetBkMode`
Brush	`GetCurrentObject`	`SelectObject`
Brush alignment origin	`GetBrushOrgEx`	`SetBrushOrgEx`
Fill mode	`GetPolyFillMode`	`SetPolyFillMode`
Stretch mode	`GetStretchBltMode`	`SetStretchBltMode`
Font	`GetCurrentObject`	`SelectObject`
Text alignment	`GetTextAlign`	`SetTextAlign`
Text justification width	————	`SetTextJustification`
Intercharacter spacing	`GetTextCharacterExtra`	`SetTextCharacterExtra`
Color palette	`GetCurrentObject`	`SelectPalette`
Text color	`GetTextColor`	`SetTextColor`
Background color	`GetBkColor`	`SetBkColor`

- A brush (like a QuickDraw pattern) for filling areas
- A brush alignment origin that controls the way the pixels of the brush align with the coordinate system
- A fill mode that determines which areas constitute the interior of complex figures such as self-intersecting polygons or regions
- A stretch mode that governs the scaling of bitmaps
- A font for drawing text
- A text alignment mode, justification width, and intercharacter spacing for positioning characters during text layout
- A color palette
- Text and background colors

In keeping with general Windows policy, the device context is not defined as a public data structure whose contents you can manipulate directly. All references to a device context are made via an opaque handle of type **`HDC`**. The GDI interface provides functions you can call to access or change specific attributes. Table 6–4 lists these access functions.

GDI Objects

If you look closely at Table 6–4, you'll see that several of the device context's attributes share the same pair of access functions, **`GetCurrentObject`** for retrieving the attribute and **`SelectObject`** for changing it. All of these attributes belong to a general category of data structures known as *GDI objects.* The following are all GDI objects:

- Bitmaps
- Paths
- Regions
- Pens
- Brushes
- Fonts
- Palettes

Each of these has its own descriptive data structure, such as **`BITMAP`** or **`LOGPEN`** ("logical pen"), along with a corresponding handle type, such as **`HBITMAP`** or **`HPEN`** (see Table 6–5). There's also a general-purpose handle type, **`HGDIOBJ`**, that embraces them all.

Table 6–5. GDI objects

Object type	Handle type	Data structure	Access selector	Stock objects
Bitmap	**HBITMAP**	**BITMAP**	**OBJ_BITMAP**	————
Path	————	————	————	————
Region	**HRGN**	**RGNDATA**	————	————
Pen	**HPEN**	**LOGPEN**	**OBJ_PEN**	**BLACK_PEN** **WHITE_PEN** **NULL_PEN**
Brush	**HBRUSH**	**LOGBRUSH**	**OBJ_BRUSH**	**BLACK_BRUSH** **DKGRAY_BRUSH** **GRAY_BRUSH** **LTGRAY_BRUSH** **WHITE_BRUSH** **NULL_BRUSH**
Font	**HFONT**	**LOGFONT**	**OBJ_FONT**	**SYSTEM_FONT** **SYSTEM_FIXED_FONT** **ANSI_VAR_FONT** **ANSI_FIXED_FONT** **OEM_FIXED_FONT** **DEVICE_DEFAULT_FONT**
Palette	**HPALETTE**	**LOGPALETTE**	**OBJ_PAL**	**DEFAULT_PALETTE**

Many of the most commonly needed GDI objects are available from the system as *stock objects:* for instance, there are stock black, white, and null (transparent) pens; stock brushes in black, white, and three shades of gray; and so on. You can request any of these standard objects by calling the Windows function **GetStockObject** with a selector identifying the particular object you want, such as **BLACK_PEN** or **LTGRAY_BRUSH**. The available stock objects of each type are listed in Table 6–5; we'll discuss them in more detail as they come up in later sections.

If none of the stock objects meet your needs, you can create your own GDI objects with the appropriate special-purpose Windows functions, such as **CreatePen**, **CreateFont**, or **CreateBitmap**. Whether you use a stock object or roll your own, once you have your hands on a GDI object, you must "select it into" a device context with the **SelectObject** function in order to use it. **SelectObject** returns a handle to the previous object being replaced; if you need to, you can use this handle to save the previous object and restore it later.

To retrieve any of the GDI objects associated with a device context, you call the function **GetCurrentObject**. This function accepts a handle to the context, along with one of the access selectors listed in the table, such as **OBJ_PEN** or **OBJ_BRUSH**, and returns a handle to the object of the requested type currently selected into the context. To get more information about the object, you can then pass this handle in turn to the function **GetObject**, along with a pointer to a buffer in memory to be filled in with the appropriate descriptive data structure, as shown in the table.

Part of the significance of GDI objects is that, even though they may be created at your request, they belong to the Graphics Device Interface itself and not to your program. This means that,

unlike other data structures your program creates in the course of its execution, GDI objects like pens and brushes are not automatically destroyed for you when your program terminates. It's your responsibility to clean up after yourself by destroying all of your GDI objects with the Windows function **DeleteObject** when you no longer need them. If you don't, they will continue to sit around clogging up memory even after you're gone. Be careful, though: this applies only to those GDI objects that you explicitly create for yourself. Unless you have a strong suicidal streak, don't ever call **DeleteObject** on a stock object, nor on any object that's still selected into a device context!

Obtaining a Device Context

Because device contexts are such large, unwieldy data structures, Windows doesn't normally keep them lying around in memory when they're not in use. Unlike a Macintosh window record, which has a complete graphics port permanently embedded within it, a Windows window typically is not permanently associated with a device context. Instead, the Windows GDI keeps a small number of anonymous device contexts available to be shared by all programs in the system. When you need to do some drawing, you normally borrow one of these "common" device contexts for as long as you need it, then release it back to the system when you're through with it, making it available for reuse by your own or some other program.

The most common reason for obtaining a device context is to update the contents of a window for which you've received a **WM_PAINT** message. The code for responding to such a message typically looks something like this:

```
VOID DoPaint (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_PAINT message.

   {
      HDC          paintContext;             // Device context for window
      PAINTSTRUCT  paintInfo;                // Information structure for painting

      paintContext = BeginPaint (thisWindow, &paintInfo);   // Get device context

         . . . paintContext . . . ;                         // Use it for drawing

      EndPaint (thisWindow, &paintInfo);                    // Release it when done

   }  /* end DoPaint */
```

The **BeginPaint** function returns a handle to a device context initialized for painting in the client area of the window to be updated. Like the equivalent Macintosh Toolbox routine **BeginUpdate**, it intersects the context's clipping region with the window's update region, so that all drawing operations will be confined within the area that needs repainting. The **PAINTSTRUCT** parameter is a pointer to a data structure that **BeginPaint** fills in with information describing the needed update. When you're finished drawing into the window, you call **EndPaint** to release the device context back to the system.

For general drawing in a window (not in response to a **WM_PAINT** message), you can use the function **GetDC** to obtain a device context for the window's client area. In this case, you use **ReleaseDC** instead of **EndPaint** to recycle the context when you're through with it:

```
theContext = GetDC (thisWindow);            // Get device context for client area

   . . . theContext . . . ;                 // Use it for drawing

ReleaseDC (thisWindow, theContext);         // Release device context
```

There's also an extended version of the **GetDC** function, **GetDCEx**, that accepts an additional word of option flags as a parameter, allowing you to exercise finer control over the context's clipping and other aspects of its behavior. Another related function, **GetWindowDC**, returns a device context for drawing the entire window, including the nonclient area (title bar, menu bar, sizing border, scroll bars, and so forth) as well as the client area:

```
theContext = GetWindowDC (thisWindow);      // Get device context for full window

   . . . theContext . . . ;                 // Use it for drawing

ReleaseDC (thisWindow, theContext);         // Release device context
```

You might use this function, for example, if you were customizing the window's appearance by intercepting the nonclient painting message **WM_NCPAINT**, which normally gets passed through to the system via the default window procedure **DefWindowProc**.

Each time you ask Windows for a common device context, it reinitializes all of the context's attributes to their default values; you must then change these settings by hand if you want to use values other than the defaults. If you're planning to do a lot of drawing in a particular environment, you can save the trouble of setting up all the attributes each time by giving the window a device context of its own, which will be saved between drawing operations instead of being returned to common storage. You do this by specifying the style option **CS_OWNDC** (listed earlier in Table 3–3) when registering the window's class. Each window of the class will then be given its own private device context, just as a Macintosh window has its own graphics port. Any settings you make to the attributes of such a context will be retained unchanged from one drawing operation to the next, even if you call **ReleaseDC** or **EndPaint** in the meantime. When you destroy the window, its private device context will automatically be destroyed along with it. Keep in mind, though, that device contexts do take up a lot of space in memory, so use this capability sparingly and only when the extra convenience outweighs the added memory usage.

Sometimes, before making changes to the attributes of a device context, you may want to save its existing drawing environment so you can restore it again later. The Windows functions **SaveDC** and **RestoreDC** allow you to do this. **SaveDC** saves a complete copy of the context's current attributes on a special stack and returns an integer reference value that you can use later to restore the attributes to their original state:

```
saveState = SaveDC (theContext);            // Save drawing environment

   . . . theContext . . . ;                 // Change attributes and draw

RestoreDC (theContext, saveState);          // Restore saved environment
```

Device Capabilities

More so than the Macintosh, Windows has always had to deal with a proliferation of different display and hardcopy devices, as well as an alphabet soup of video interfaces like CGA (Color

Graphics Adapter), EGA (Enhanced Graphics Adapter), VGA (Video Graphics Array), and XGA (Extended Graphics Adapter). These devices have a wide range of graphical capabilities, from a 640-by-200-pixel screen to 1600-by-1200, from monochrome to 32-bit color, from square pixels to rectangular ones at aspect ratios of 1:2 or 5:12. Some devices have special hardware support for operations like clipping, curve drawing, or character rotation. Needless to say, the same drawing operations can come out looking very different depending on the characteristics of the device you draw them on.

Luckily, Windows includes a number of features and services to help you overcome the effects of such device dependencies. Graphical system resources such as cursors and icons, for instance, carry alternative definitions for use on different display devices; when you request such a resource, Windows automatically chooses the definition best suited to the available screen. As we'll see, the Windows GDI offers you a choice of device-dependent and -independent ccordinate systems to do your drawing in. And the GDI function **`GetDeviceCaps`** returns information on the capabilities of the available graphics device. Taking this information into account in your graphical operations can help you make your programs "parametrically device-independent."

`GetDeviceCaps` takes two parameters: a handle to a device context and an integer selector identifying the specific device information you're requesting (see Table 6–6). If the device context is for a window on the screen, the function returns information about the characteristics of the screen; if it's a printer device context (like a Macintosh printing port), the function returns information about the printer.

If all you're doing is requesting device information, you can use a special form of device context called an *information device context,* which dispenses with many of the data structures needed to support drawing operations. This makes it more efficient than an ordinary device context for retrieving device information, but also means that you can't use it for actual drawing. The Windows function **`CreateIC`** creates an information context for a specified device and driver. Because such a context doesn't come from the pool of common device contexts, you use a different function to dispose of it when you're finished with it: **`DeleteDC`** instead of the more usual **`ReleaseDC`**.

Table 6–6. `GetDeviceCaps` selectors

Selector	Meaning
DRIVERVERSION	Device driver version
TECHNOLOGY	Device type
VREFRESH	Vertical refresh rate in hertz
HORZSIZE	Screen width in millimeters
VERTSIZE	Screen height in millimters
HORZRES	Screen width in pixels
VERTRES	Screen height in pixels
DESKTOPHORZRES	Width of virtual desktop in pixels
DESKTOPVERTRES	Height of virtual desktop in pixels
LOGPIXELSX	Horizontal dot pitch in pixels per inch
LOGPIXELSY	Vertical dot pitch in pixels per inch
ASPECTX	Relative pixel width
ASPECTY	Relative pixel height
ASPECTXY	Relative pixel diagonal
BLTALIGNMENT	Preferred horizontal alignment
BITSPIXEL	Color depth in bits per pixel
PLANES	Number of color planes
NUMCOLORS	Color-table capacity
SIZEPALETTE	Number of entries in system palette
NUMRESERVED	Number of reserved entries in system palette
COLORRES	Color resolution in bits per pixel
PHYSICALWIDTH	Physical width in device units (hardcopy devices only)
PHYSICALHEIGHT	Physical height in device units (hardcopy devices only)
PHYSICALOFFSETX	Horizontal margin of printable area (hardcopy devices only)
PHYSICALOFFSETY	Vertical margin of printable area (hardcopy devices only)
SCALINGFACTORX	Horizontal scaling factor (hardcopy devices only)
SCALINGFACTORY	Vertical scaling factor (hardcopy devices only)
NUMPENS	Number of built-in pens
NUMBRUSHES	Number of built-in brushes
NUMFONTS	Number of built-in fonts
RASTERCAPS	Raster capabilities (flag word)
LINECAPS	Line-drawing capabilities (flag word)
CURVECAPS	Curve-drawing capabilities (flag word)
POLYGONALCAPS	Polygon-drawing capabilities (flag word)
TEXTCAPS	Text capabilities (flag word)
CLIPCAPS	Clipping capability (flag)

Bitmaps and Memory Device Contexts

Except for information contexts, all of the device contexts that we've looked at so far have been *display device contexts* designed for drawing on the screen. Like a Macintosh graphics port, however, a Windows device context can also be set up to draw into an offscreen bitmap. This type of device context is known as a *memory device context.* In order to understand how they work, we'll first have to talk a bit about bitmaps.

Early versions of the Windows system worked with *device-dependent bitmaps* (*DDBs*) intended for use on a particular display device—usually whatever device was actually available on the user's machine at run time. Because bitmaps of this kind are tied to the specific characteristics of the target device, such as aspect ratios and color capabilities, they can't generally be used on another device without distortion. Although Win32 still supports them for the sake of compatibility, new programs being developed or ported from other platforms should use the new form of *device-independent bitmaps* (*DIBs*) instead.

A device-independent bitmap begins with a header of type **`BITMAPINFOHEADER`** (Listing 6–2) containing parametric information about the bitmap's dimensions and color format. This is followed by a color table defining the color values used in the bitmap's image, then by an array of bytes containing the pixels of the image itself. You can access or change the contents of the color table with the Windows functions **`GetDIBColorTable`** and **`SetDIBColorTable`**, the pixel image with **`GetDIBits`** and **`SetDIBits`**, or individual pixels with **`GetPixel`** and **`SetPixel`**.

The pixel array holding the image differs from those on the Macintosh in two important ways that you should be aware of. First, in monochrome bit images, Windows uses the opposite bit polarity from the Macintosh: **`0`** for a pixel of the foreground color (normally black), **`1`** for the background color (normally white). Second, the image's scan lines are normally stored from bottom to top rather than from top to bottom. (A negative value in the header's **`biHeight`** field indicates that the lines go from top to bottom instead.) The image may optionally be compressed to save space, using a 4-bit or 8-bit run-length-encoding compression format: this is indicated by an integer code in the **`biCompression`** field, either **`BI_RLE4`** or **`BI_RLE8`**. The code **`BI_RGB`** denotes an ordinary, uncompressed bit image.

Listing 6–2. Device-independent bitmap header

```
typedef struct tagBITMAPINFOHEADER
   {
      DWORD  biSize;               // Size of header in bytes; offset to color table
      LONG   biWidth;              // Width in pixels
      LONG   biHeight;             // Height in pixels
      WORD   biPlanes;             // Number of color planes
      WORD   biBitCount;           // Color depth in bits per pixel
      DWORD  biCompression;        // Compression format
      DWORD  biSizeImage;          // Size of image in bytes
      LONG   biXPelsPerMeter;      // Horizontal dot resolution in pixels per meter
      LONG   biYPelsPerMeter;      // Vertical dot resolution in pixels per meter
      DWORD  biClrUsed;            // Number of colors used
      DWORD  biClrImportant        // Number of "important" colors

   }  BITMAPINFOHEADER;
```

Recall that bitmaps, whether device-dependent or device-independent, are one of the seven types of GDI object. This means that you can select a bitmap into a device context with the **SelectObject** function. The bitmap then becomes the context's "canvas," or drawing surface: all drawing operations performed in that context will affect the contents of the bitmap's image, rather than the display screen. The context that you use for this purpose must be a memory device context instead of a display device context.

The Windows function for creating a memory device context is named **CreateCompatibleDC**, because such a context is always created to be compatible with the graphical characteristics of a specified device. This function accepts a handle to an existing device context, such as a display context associated with a window, and creates a new memory device context compatible with the same device as the designated context. If you leave the parameter null, the new context will be made compatible with the current display device by default. Because the memory device context, like an information device context, is created from scratch rather than borrowed from the pool of common device contexts, you use the **DeleteDC** function, rather than **ReleaseDC**, to dispose of it when you're through with it.

When first created, a memory device context has a dummy bitmap just one pixel wide by one high. You can then replace this with an existing bitmap, load a bitmap from a resource with **LoadBitmap**, or use **CreateCompatibleBitmap** to create a brand-new bitmap compatible with the given context. In any case, you use **SelectObject** to select the new bitmap into the device context. You can then proceed to draw into the bitmap with any GDI drawing operations.

Clipping

You can set a clipping boundary for a device context with the Windows function **SelectClipRgn**. This makes a copy of a specified region and selects it into the device context as a GDI object. All subsequent drawing operations in the context will be confined within the interior of this region. The original copy of the region is unaffected by the operation, and can safely be deleted after you've copied it into the device context. The **GetClipRgn** function returns a context's clipping region; **GetClipBox** returns its bounding box. Calling **SelectClipRgn** with a null region parameter removes any existing clipping region from a device context.

An extended form of the function, **ExtSelectClipRgn**, allows you to combine another region with a context's existing clipping region: by supplying an appropriate mode parameter, you can form the union, intersection, nonintersection, or difference of the two regions. You can also subtract a rectangle from the clipping region with **ExcludeClipRect**, intersect it with a rectangle with **IntersectClipRect**, move it with **OffsetClipRgn**, test whether it includes a given point with **PtVisible**, or test whether it intersects a given rectangle with **RectVisible**.

For greater generality, you may want to use a path instead of a region for your clipping boundary. You begin by defining the path into the desired device context, using any sequence of GDI drawing operations bracketed by calls to **BeginPath** and **EndPath**. You can then convert this path into a region and make it the context's clipping region

```
pathRegion = PathToRegion(theContext); // Convert context's path to region
SelectClipRgn(theContext, pathRegion); // Make it the clipping region
```

or you can use the function **SelectClipPath** to combine it in various ways with the context's existing clipping region:

```
SelectClipPath(theContext, RGN_COPY);  // Copy path to clipping region
SelectClipPath(theContext, RGN_AND);   // Intersect path with clipping region
SelectClipPath(theContext, RGN_OR);    // Unite path with clipping region
SelectClipPath(theContext, RGN_XOR);   // Form nonintersection of path and region
SelectClipPath(theContext, RGN_DIFF);  // Subtract path from clipping region
```

Coordinate Systems

One of the attributes of a device context is a *mapping mode* that defines the logical coordinate system in which it does its drawing. The mapping mode establishes the correspondence between the context's logical coordinates and the physical coordinate system of the target device. The Windows function **SetMapMode se**ts a device context's mapping mode; **GetMapMode** retrieves it; **LPtoDP** and **DPtoLP** convert a point between logical and physical (device) coordinates according to the context's current mapping mode. Table 6–7 shows the possible modes that are available.

Mapping Modes

In the default mapping mode, **MM_TEXT**, coordinates increase from left to right and from top to bottom—the natural direction for English text and also the same one used on the Macintosh. **MM-TEXT** is also the only mapping mode whose coordinates are expressed directly in pixels, which means that its coordinate measurements are device-dependent. If you draw a rectangle 96 pixels wide by 96 high in **MM_TEXT** mode, it will appear as a one-inch square on a VGA screen (whose aspect ratio is 1:1), but it will be an inch wide by one-and-a-third high on an EGA, an inch wide by two high on a CGA, and only about one-quarter inch square on a 300-dot-per-inch laser printer. If you want the figure to look more or less the same on any output device, you have to retrieve the device characteristics with **GetDevCaps** and adjust for them parametrically.

Most of the other mapping modes express their coordinates in device-independent units. One of them, **MM_TWIPS**, uses the same axis orientation as **MM_TEXT**, from left to right and top to

Table 6–7. Mapping modes

Mode	Unit	Positive X	Positive Y	Aspect ratio
MM_TEXT	Pixel	Right	Down	Device-dependent
MM_TWIPS	1/20 pt (1/1440 in)	Right	Down	Uniform
MM_LOENGLISH	0.01 in	Right	Up	Uniform
MM_HIENGLISH	0.001 in	Right	Up	Uniform
MM_LOMETRIC	0.1 mm	Right	Up	Uniform
MM_HIMETRIC	0.01 mm	Right	Up	Uniform
MM_ISOTROPIC	Application-defined	Application-defined	Application-defined	Uniform
MM_ANISOTROPIC	Application-defined	Application-defined	Application-defined	Application-defined

bottom, but measures its coordinates in "twips"—twentieths of a printer's point—rather than pixels. Windows takes a point to be exactly 1/72 of an inch (though in the traditional print world this is actually a slight approximation), so there are 1440 twips to the inch. The Windows GDI takes the responsibility for converting these device-independent units into the available dot resolution and aspect ratio of a given physical device.

The next four modes are also device-independent and, unlike the text-based modes, have their vertical axes oriented in the conventional mathematical direction, from bottom to top. Two of them use English units of measure (inches) and two use metric units. Each pair consists of a lower- and a higher-resolution mode in which the scale of the axes differs by a factor of ten: from **MM_LOENGLISH**, in which the unit of measure is a hundredth of an inch, to **MM_HIENGLISH**, which measures in thousandths of an inch, and from **MM_LOMETRIC** (tenths of a millimeter) to **MM_HIMETRIC** (hundredths of a millimeter). In all of these cases, the scale is uniform along both axes: that is, the horizontal units are the same size as the vertical ones.

The final two modes allow you to establish your own correspondence between logical and physical coordinates. You do this by specifying a pair of reference rectangles, a *window* in logical space and a *viewport* in physical (device) space. You can set the rectangles for a given device context by using the Windows functions **SetWindowOrgEx** and **SetWindowExtEx** for the origin and extent of the logical window and **SetViewportOrgEx** and **SetViewportExtEx** for those of the physical viewport. (The suffix **Ex** indicates that these are "extended" versions of these functions that supersede the originals used in earlier versions of the Windows system.) Similarly, the functions for retrieving the reference rectangles from a device context are **GetWindowOrgEx**, **GetWindowExtEx**, **GetViewportOrgEx**, and **GetViewportExtEx**.

The ratio between the widths of the two reference rectangles establishes the horizontal scaling factor between logical and physical coordinates, and similarly in the vertical direction. Physical coordinates are understood to increase in a textlike direction, from left to right and top to bottom; a negative ratio in the rectangles' extents in either dimension reverses the sense of the corresponding axis in logical coordinate space. In **MM_ISOTROPIC** mode, the scale factors along the two axes are constrained to be equal; in **MM_ANISOTROPIC** mode, they can differ. The Windows functions **ScaleWindowExtEx** and **ScaleViewportExtEx** adjust the extents of the window and viewport rectangles, respectively, by a specified horizontal and vertical scale factor.

The offset between the origins of the window and viewport rectangles determines the location of the logical coordinate origin in physical device space. You can adjust the relationship between the two rectangles with the functions **OffsetWindowOrgEx** and **OffsetViewportOrgEx**. This correspondence is further affected by the type of device context and the way in which it was obtained. Depending on the type of context and the specific operation being performed, the coordinates may be expressed relative to the top-left corner of a window's client area (*client coordinates*), of the entire window, including both client and nonclient areas (*window* or *whole-window* coordinates), of the display screen (*screen coordinates*), or of the printer page (*page coordinates*). The functions **ClientToScreen** and **ScreenToClient** convert a point between client and screen coordinates. Another function, **MapWindowPoints**, converts an array of points from one window's coordinate system to that of another.

World Transforms

In addition to the simple translation and scaling performed by the mapping mode, Win32 allows you to define a further, optional linear transformation for any device context. If present, this *world transform* is applied to all logical coordinates specified in the context before they are mapped by the mapping mode. The world transform is defined by a 3-by-3 linear transformation matrix of the form

```
| eM11   eM12   0 |
| eM21   eM22   0 |
| eDx    eDy    1 |
```

where the coefficients **eM11**, **eM12**, **eM21**, **eM22**, **eDx**, and **eDy** are specified in a structure of type **XFORM** (Listing 6–3). By choosing suitable values for these coefficients, you can achieve not only translation and scaling, but also more sophisticated transformations such as rotation, reflection, and shear. We won't go into the details here, but you can find all the information you need in the *Win32 Programmer's Reference.*

World transforms are a new feature introduced in Win32; they didn't exist in earlier versions of the Windows system. For compatibility with those earlier versions, Win32 device contexts have a *graphics mode* that controls whether the world-transform feature is available. When the context is first created, it's set to *compatible mode.* This gives it the same behavior as in earlier versions of the system: the only transformation applied between logical and physical coordinates is that defined by the mapping mode. The context's world transform is initialized to a simple identity transformation that just passes all logical coordinates on unchanged to the mapping mode:

```
| 1   0   0 |
| 0   1   0 |
| 0   0   1 |
```

If you want to replace this with some other, more interesting transformation, you must first change the graphics mode from compatible to advanced, using the Windows function **SetGraphicsMode**. You can then specify a new transformation matrix with **SetWorldTransform**, or apply a further transformation by using **ModifyWorldTransform** to multiply the existing matrix by another, on either the left or the right. The **CombineTransform** function multiplies two transformation matrices to produce a third, without reference to any particular device context. You can retrieve a context's current transformation matrix with **GetWorldTransform** or find out its current graphcis mode with **GetGraphicsMode**.

Listing 6–3. World transform coefficient structure

```
typedef struct _XFORM
   {
      FLOAT  eM11;
      FLOAT  eM12;
      FLOAT  eM21;
      FLOAT  eM22;
      FLOAT  eDx;
      FLOAT  eDy;

   }  XFORM;
```

Line Drawing

Just as on the Macintosh, you draw lines in Windows by moving a graphics pen around from point to point on the coordinate plane. Each device context has its own pen, a GDI object that is associated with the context using the **`SelectObject`** function. The context maintains a current pen position, which is updated as you use the pen to perform drawing operations. In addition to straight lines, the Windows GDI includes facilities for drawing regular and irregular curves and for combining line and curve segments to form more complex figures in a single operation.

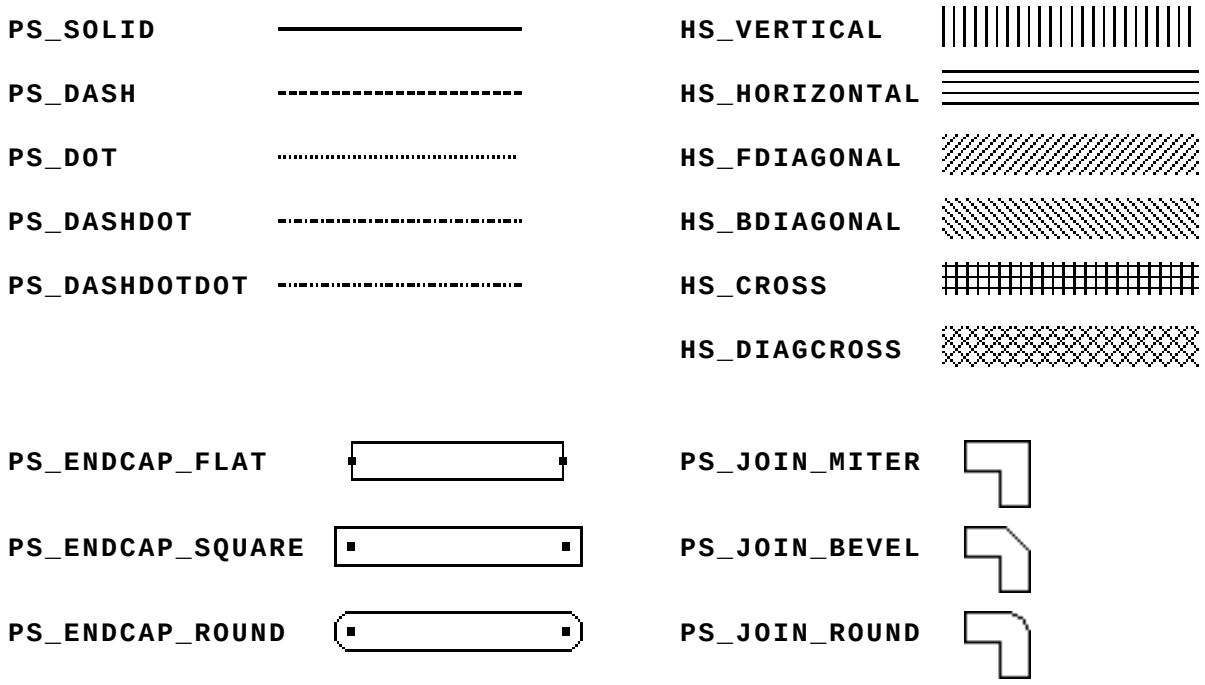
Pens

Pens come in two types: *cosmetic pens* and *geometric pens.* The simpler of the two, a cosmetic pen, is always exactly one pixel wide, regardless of the device it's used on. This makes a cosmetic pen draw very quickly and efficiently, but the resulting lines scale with the resolution of the device, appearing thinner at higher resolutions and thicker at lower resolutions. Geometric pens are slower to use, but offer a more flexible range of options and capabilities.

In its full generality, a pen has the following attributes:

- A width
- A color
- A line style
- A pattern
- A hatch style
- An end-cap style
- A join style

Figure 6–2. Pen styles



A cosmetic pen has only the first three of these (width, color, and line style), and as noted, its width is always limited to exactly one pixel; a geometric pen can be set to any desired width. Furthermore, the width of a geometric pen is not expressed in pixels, but in logical units relative to the context's current mapping mode, independent of the device's dot resolution. The last four attributes (pattern, hatch, end cap, and join) belong to geometric pens only. Figure 6–2 illustrates the various line, hatch, end-cap, and join styles available.

For the most straightforward uses, Windows provides three stock cosmetic pens—black, white, and null (transparent)—which you can retrieve with the function **`GetStockObject`**. If you need something fancier, you can create your own cosmetic pen by passing the desired width, color, and line style directly to the Windows function **`CreatePen`** or by storing them into a structure of type **`LOGPEN`** ("logical pen") and passing the structure to **`CreatePenIndirect`**. The more general, extended function **`ExtCreatePen`** can create both cosmetic and geometric pens. All of these pen-creation functions take a style parameter that's a combination of the option flags listed in Table 6–8, and return a handle of type **`HPEN`**. (Only the line styles are valid for **`CreatePen`** and **`CreatePenIndirect`**; **`ExtCreatePen`** accepts them all.) **`ExtCreatePen`** also accepts a brush as a parameter, from which it takes the pen's color, pattern, and hatch style; we'll look at these attributes more closely later when we talk about brushes. In any case, once you have a pen, you must select it into a device context with **`SelectObject`** in order to draw with it.

Table 6–8. Pen style options

Style name	Meaning
PS_COSMETIC	Cosmetic pen
PS_GEOMETRIC	Geometric pen
PS_SOLID	Solid line
PS_DASH	Dashed line
PS_DOT	Dotted line
PS_DASHDOT	Alternating dashes and dots
PS_DASHDOTDOT	Alternating dashes and double dots
PS_ALTERNATE	Alternating pixels
PS_INSIDEFRAME	Solid line confined within figure boundary
PS_NULL	Invisible
PS_USERSTYLE	Application-defined line style
PS_ENDCAP_FLAT	Flat end caps
PS_ENDCAP_SQUARE	Square end caps
PS_ENDCAP_ROUND	Round end caps
PS_JOIN_MITER	Mitered joins
PS_JOIN_BEVEL	Beveled joins
PS_JOIN_ROUND	Round joins

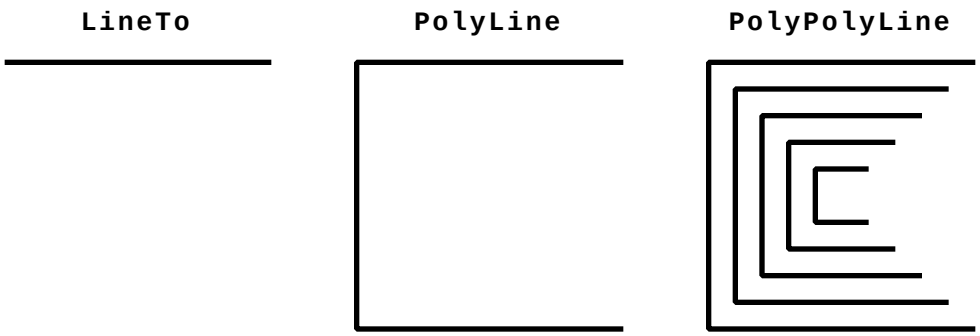
Straight Lines

Every device context maintains a current pen position, which is a point in the logical coordinate space defined by the context's mapping mode. In a brand-new device context, the pen position is initialized to the coordinate origin (**0**, **0**); you can then move it to any other coordinates you like before beginning a drawing operation. The original Windows function for moving the pen, **MoveTo**, is now considered obsolete; it has been replaced in Win32 with an extended version, **MoveToEx**. The difference between the two is that the new version, in addition to moving the pen to a new position, returns the coordinates of its previous position via an output parameter. This allows you, if appropriate, to save and restore the pen position so that it is left unaffected by your drawing operation:

```
MoveToEx (theContext, newX, newY, &oldPoint);   // Move pen and save old position
   oldX = oldPoint.x;                           // Extract previous coordinates
   oldY = oldPoint.y;                           //    from returned point
   . . . ;                                      // Draw something
MoveToEx (theContext, oldX, oldY, NULL);        // Restore previous position
```

The **LineTo** function moves the pen from its current position to a specified new position, drawing as it goes. The result is to draw a straight line from one point to the other, using the attributes (width, color, pattern, and so on) of the context's currently selected pen. The starting and ending points are always expressed in absolute logical coordinates: there are no equivalents to the Macintosh **Move** and **Line** routines, which move the pen by a relative offset from its

Figure 6–3. Line-drawing operations



current position. One other difference to watch out for is that in Windows, the resulting line always includes the starting point but *excludes* the ending point. Windows refers to this as "inclusive-exclusive" line drawing.

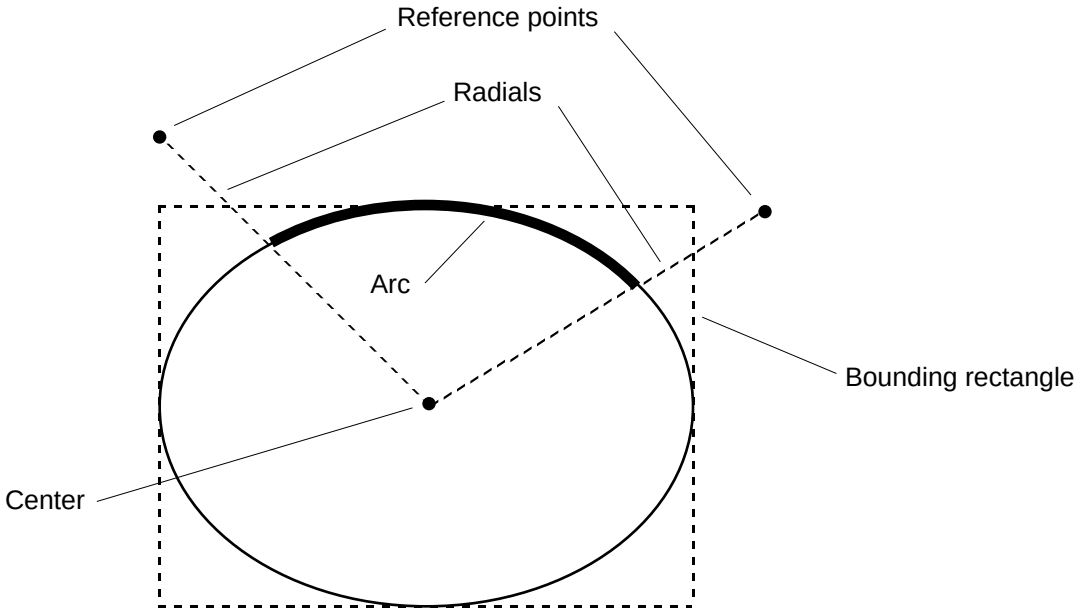
A pair of functions called **`Polyline`** and **`PolylineTo`** accept an array of endpoints and draw a series of connected lines from one to the next, beginning at the current pen position. The difference between the two functions is that **`PolylineTo`** leaves the pen positioned at the final ending point of the figure, while **`Polyline`** leaves it back at the original starting point. Neither function closes the figure by automatically filling in a concluding line from the end back to the beginning. An even higher-level function, **`PolyPolyline`**, draws multiple polylines in a single operation, allowing you to construct complex line figures like the one shown in Figure 6–3.

Curves

The Windows GDI includes a variety of facilities for drawing curves, both regular (circular and elliptical arcs) and irregular (Bézier curves). The basic **`Arc`** function draws a regular arc forming part of a circle or ellipse, using the pen currently selected into the designated device context. The complete figure is defined by a bounding rectangle, just like an oval on the Macintosh. The portion to be drawn is delimited by a pair of *radials* extending from the center of the figure to two reference points that you supply as parameters. The endpoints of the arc are the points where the radials cross the circumference of the circle or ellipse (see Figure 6–4).

By default, the arc runs counterclockwise around the circumference from the starting point to the ending point; if you wish, you can reverse this drawing direction with the Windows function **`SetArcDirection`**. The drawing direction is significant because, just as when drawing a straight line, the drawing operation includes the arc's starting point but excludes its ending point. The simple **`Arc`** function just draws the specified arc, without affecting the pen position. An alternate form, **`ArcTo`**, draws not only the arc itself, but also a straight line from the current pen position to the arc's starting point; after it's through, it leaves the pen positioned at the end of the arc.

Figure 6–4. Defining an arc with **`Arc`** or **`ArcTo`**



A function called **`AngleArc`** uses a different method to specify the arc to be drawn. Instead of an ellipse defined by its bounding rectangle, **`AngleArc`** operates on a circle defined by its center and radius. The extent of the arc is given by a *start angle* and a *sweep angle*, rather than by a pair of radials (see Figure 6–5). The angles are expressed in degrees and measured counterclockwise around the circle: the start angle begins at the horizontal (*x*) axis, and the sweep angle begins at the end of the start angle. Like **`ArcTo`**, **`AngleArc`** begins by drawing a straight line from the original pen position to the beginning of the arc, followed by the arc itself. The pen is left positioned at the end of the arc. Although the operation is nominally limited to circular arcs, you

Figure 6–5. Defining an arc with **`AngleArc`**

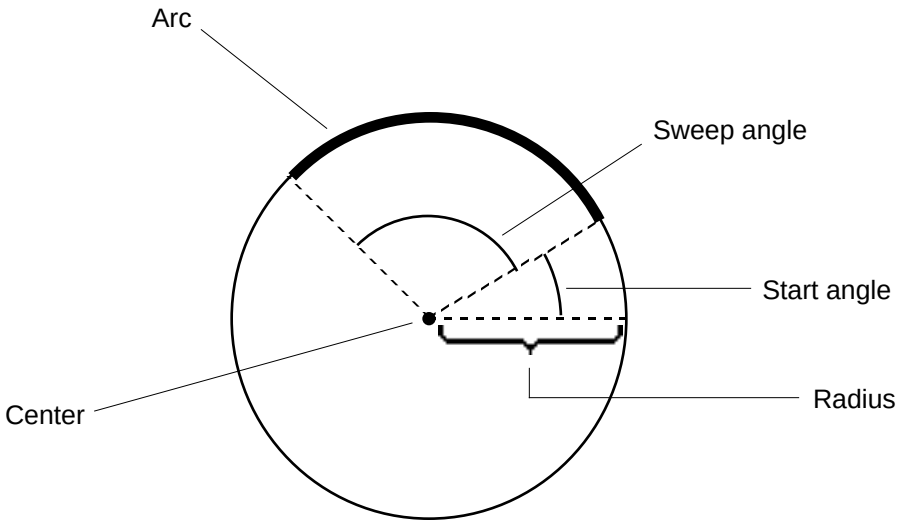
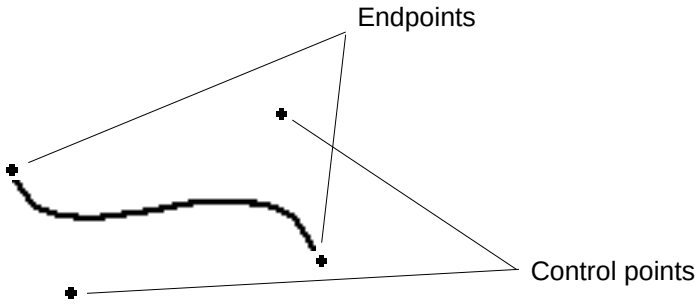


Figure 6–6. Defining a Bézier curve



can actually produce elliptical results by using a suitable mapping mode or world transform.

The most general form of curve you can draw is a cubic spline (Bézier) curve. Such a curve is defined by a series of four points: two endpoints and two control points that mark where the curve changes direction (see Figure 6–5). The Windows function **`PolyBezier`** accepts an array of points and draws a series of Bézier curves, each beginning from the ending point of the one before. As with the arc functions, the straight **`PolyBezier`** function leaves the device context's pen position unchanged, while **`PolyBezierTo`** leaves it at the end of the last Bézier segment drawn. Finally, a function named **`PolyDraw`** allows you to string together both straight and curved segments to form line figures of arbitrary complexity. Table 6–9 summarizes the available line-drawing functions.

Table 6–9. Line-drawing functions

Function	Purpose	Pen Position
`MoveToEx`	Move pen to new position	Updated
`LineTo`	Draw straight line	Updated
`Polyline`	Draw series of connected straight lines	Unchanged
`PolylineTo`	Draw series of connected straight lines	Updated
`PolyPolyline`	Draw multiple series of connected straight lines	Unchanged
`Arc`	Draw circular or elliptical arc between two radials	Unchanged
`ArcTo`	Draw circular or elliptical arc between two radials	Updated
`AngleArc`	Draw circular arc defined by two angles	Updated
`GetArcDirection`	Get arc direction (clockwise or counterclockwise)	————
`SetArcDirection`	Set arc direction (clockwise or counterclockwise)	————
`PolyBezier`	Draw series of Bézier curves	Unchanged
`PolyBezierTo`	Draw series of Bézier curves	Updated
`PolyDraw`	Draw series of straight lines and Bézier curves	Updated

Shape Drawing

Like QuickDraw on the Macintosh, the Windows GDI allows you to draw a variety of geometric shapes. There are GDI functions for drawing rectangles with both square and rounded corners, ellipses (called ovals on the Macintosh), elliptical sectors (called wedges on the Macintosh) and segments, polygons, and regions and paths of arbitrary shape. As always, all shape-drawing operations must take place with reference to a device context. The general drawing operation for each shape uses the context's current pen to draw the outline of the shape and the current brush to fill the interior. The results are also governed by the context's *drawing mode*, which specifies how the pixels of the pen or brush combine with those already existing on the screen or in the target bitmap. We've already talked about pens; now let's look at brushes and modes, then see how they're used in shape-drawing operations.

Brushes

A *brush* is the Windows equivalent of a Macintosh pattern: an 8-by-8-pixel "tile" that can be endlessly repeated to fill an area in a graphical image. The simplest type of brush is a *solid brush*, representing a single uniform color to be painted into the filled area. You can create one by passing the desired color to the Windows function **`CreateSolidBrush`**, receiving back a handle of type **`HBRUSH`**. Logically, the brush represents the requested color exactly; but when you select it into a device context with **`SelectObject`**, it may turn out that the exact color isn't available on the given physical device. In that case, GDI will either replace it with the nearest available matching color or approximate it by constructing a dithered pattern out of two or more available colors. This process happens automatically, and you have no direct control over the dithering algorithm.

As we've already seen earlier in Table 6–5, Windows has predefined *stock brushes* available in black, white, and three shades of gray. You can retrieve any of these with the **`GetStockObject`** function and select it into a device context with **`SelectObject`**. Thre's also a null or "hollow" brush, which doesn't paint anything at all: this is useful, for instance, when you want to frame a shape (draw just its outline without affecting the interior).

In addition to these standard brushes, you can also create a solid brush in any of the 21 *system colors.* These represent the user's color preferences for various elements of the Windows user interface (window title bars, menu backgrounds, and so forth), which the user can customize to individual taste with the Display control panel. To create a brush representing any of the system colors, you simply add **`1`** to the corresponding selector, listed in Table 6–10, and typecast it into a brush handle. We've already seen an example of this in our NullApp example program of Chapter 3 (Listing 3–2), where we used the statement

```
bkBrush = HBRUSH(COLOR_WINDOW + 1);
```

to create a brush for painting a window's background in the user's currently selected background color.

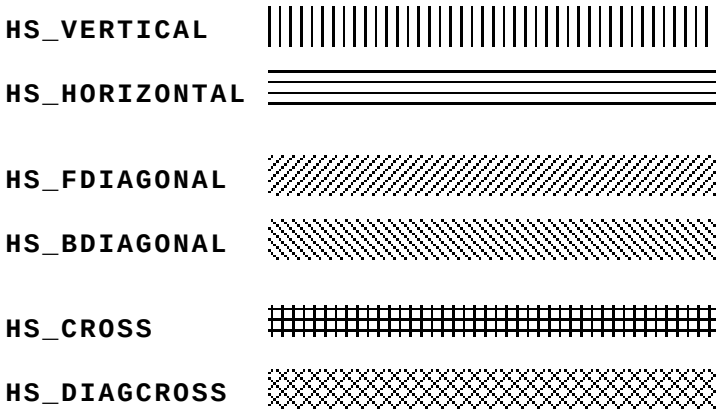
Table 6–10. System color selectors

Selector	Meaning
COLOR_BACKGROUND	Desktop background
COLOR_WINDOWFRAME	Window frame
COLOR_SHADOW	Window shadow
COLOR_ACTIVECAPTION	Title bar of active window
COLOR_CAPTIONTEXT	Title text of active window
COLOR_ACTIVEBORDER	Sizing border of active window
COLOR_INACTIVECAPTION	Title bar of inactive window
COLOR_INACTIVECAPTIONTEXT	Title text of inactive window
COLOR_INACTIVEBORDER	Sizing border of inactive window
COLOR_WINDOW	Window background
COLOR_APPWORKSPACE	Background of multiple-document-interface (MDI) window
COLOR_WINDOWTEXT	Text in window
COLOR_HIGHLIGHTTEXT	Selected text
COLOR_HIGHLIGHT	Background of selected text
COLOR_SCROLLBAR	Shaft of scroll bar
COLOR_BTNFACE	Face of pushbutton
COLOR_BTNSHADOW	Edge shadow of pushbutton
COLOR_BTNTEXT	Text of pushbutton
COLOR_MENU	Menu background
COLOR_MENUTEXT	Text of menu item
COLOR_GRAYTEXT	Text of disabled control or menu item

The second type of brush you can create is a *hatch brush* in one of six predefined hatch styles (see Figure 6–7). You identify the desired style by passing one of the selector constants in Table 6–11 to the Windows function **CreateHatchBrush**, along with an RGB color for the brush's foreground pixels (those shown in black in the figure). What happens to the background pixels depends on the *background mode* of the device context in which you use the brush. If the background mode is **OPAQUE**, the brush will use the context's background color to paint its background pixels; if the mode is **TRANSPARENT**, it will leave existing pixels of the target bitmap unchanged, allowing them to "show through" the background.

The last type of brush, a *pattern brush,* is the one that corresponds most closely to a QuickDraw pattern. The contents of the brush are defined by a bitmap that you pass as a parameter to **CreatePatternBrush** (if it's a device-dependent bitmap) or to **CreateDIBPatternBrush** (if it's a device-independent one). The bitmap is expected to be eight pixels wide by eight high; if it's bigger, only the first eight rows and eight columns in its

Figure 6–7. Standard hatch styles



top-left corner will be used. If the bitmap carries its own color information, the brush will use those colors; if it's a monochrome bitmap, the brush will use the text and background colors from the device context for its foreground and background pixels.

One more point to be aware of when using brushes is the alignment problem: two adjacent areas filled with the same brush (for instance, two child windows using the same parent's background brush) may not line up properly if the brush is aligned to different coordinate origins. To avoid unsightly seams, you may need to adjust the brush alignment for one area or the other with the Windows function **`SetBrushOrgEx`** (an extended version of an older function named **`SetBrushOrg`**). By default, the brush's top-left corner lines up with the origin of the device context's logical coordinate system; **`SetBrushOrgEx`** allows you to change this by designating a different pixel within the brush to coincide with the coordinate origin. The function returns the previous alignment coordinates through an output parameter, allowing you to save and restore them if need be. The matching function **`GetBrushOrgEx`** returns the brush's current alignment coordinates.

Table 6–11. Hatch selectors

Selector	Meaning
`HS_VERTICAL`	Vertical hatch
`HS_HORIZONTAL`	Horizontal hatch
`HS_FDIAGONAL`	Forward diagonal hatch (southwest to northeast)
`HS_BDIAGONAL`	Backward diagonal hatch (northwest to southeast)
`HS_CROSS`	Horizontal and vertical crosshatch
`HS_DIAGCROSS`	Diagonal crosshatch

Drawing Modes

Like the Macintosh, Windows has a *drawing mode* that controls the way pixels from a pen or brush combine with the existing pixels of the screen or bitmap into which they're drawing. Technically speaking, a drawing mode is called a *raster operation*, or ROP—specifically, a *binary raster operation*, or ROP2, because it operates on two operands: a source pixel from the pen or brush and a destination pixel from the screen or bitmap.

Since there are four possible combinations of source and destination pixels, there are $2^4 = 16$ combinations of results, each constituting a different raster operation, or drawing mode. Each of the sixteen has a name, as shown in Table 6–12. Some of these operations are more useful than others, of course, and there are some that you'll probably never use—but they're available in case you need them. The Windows function **`SetROP2`** sets one of them as a device context's drawing mode; **`GetROP2`** returns the current drawing mode. The default drawing mode is **`R2_COPYPEN`**, which simply copies the source pixels of the pen or brush to the destination, without reference to what was already there.

Note that the bit results shown in the table apply to a monochrome pen or brush drawing into a monochrome destination. When these same operations are applied to color bitmaps at pixel depths greater than 1, the results can be unpredictable or nonintuitive. Bear in mind also that Windows bitmaps use the opposite bit polarity from those on the Macintosh: **`0`** for a black pixel, **`1`** for white. As a result, the terms "mask" and "merge" have opposite meanings from what you might expect: masking out white pixels is the same as merging in black ones, and vice versa.

Table 6–12. Drawing modes

Mode	Pen pixel: / Screen pixel:	1 / 1	1 / 0	0 / 1	0 / 0	Description
`R2_COPYPEN`		1	1	0	0	Copy pen to screen
`R2_NOTCOPYPEN`		0	0	1	1	Copy pen to screen and invert
`R2_MERGEPEN`		1	1	1	0	Merge pen with screen
`R2_MERGEPENNOT`		1	1	0	1	Merge pen with inverse of screen
`R2_MERGENOTPEN`		1	0	1	1	Invert pen and merge with screen
`R2_NOTMERGEPEN`		0	0	0	1	Merge pen with screen and invert
`R2_MASKPEN`		1	0	0	0	Mask pen with screen
`R2_MASKPENNOT`		0	1	0	0	Mask pen with inverse of screen
`R2_MASKNOTPEN`		0	0	1	0	Invert pen and mask with screen
`R2_NOTMASKPEN`		0	1	1	1	Mask pen with screen and invert
`R2_NOT`		0	1	0	1	Invert screen
`R2_XORPEN`		0	1	1	0	Selectively invert screen
`R2_NOTXORPEN`		1	0	0	1	Invert pen and selectively invert screen
`R2_BLACK`		0	0	0	0	Paint solid black
`R2_WHITE`		1	1	1	1	Paint solid white
`R2_NOP`		1	0	1	0	Leave screen unchanged

Shapes

Figure 6–8 shows the variety of shapes that can be drawn with GDI operations:

- Rectangles
- Rounded-corner rectangles
- Ellipses
- Elliptical pie sectors and chord segments
- Polygons
- Regions and paths

Most of these are available on the Macintosh as well, though sometimes by different names: ellipses are called ovals on the Macintosh and pie sectors are called wedges, while chord segments don't exist at all. The way you define the various shapes is also pretty much the same as on the Macintosh: you specify an ellipse by giving its bounding rectangle, for instance, and a rounded-corner rectangle by giving the coordinates of the rectangle itself along with the width and height of the elliptical corners. Sectors and segments are defined in the same way we've already seen for arcs, by giving the bounding rectangle of the full ellipse along with two reference points that determine a pair of radials originating at the ellipse's center. The arc then extends counterclockwise from the point where the circumference of the ellipse crosses the first radial to

Figure 6–8. GDI shapes

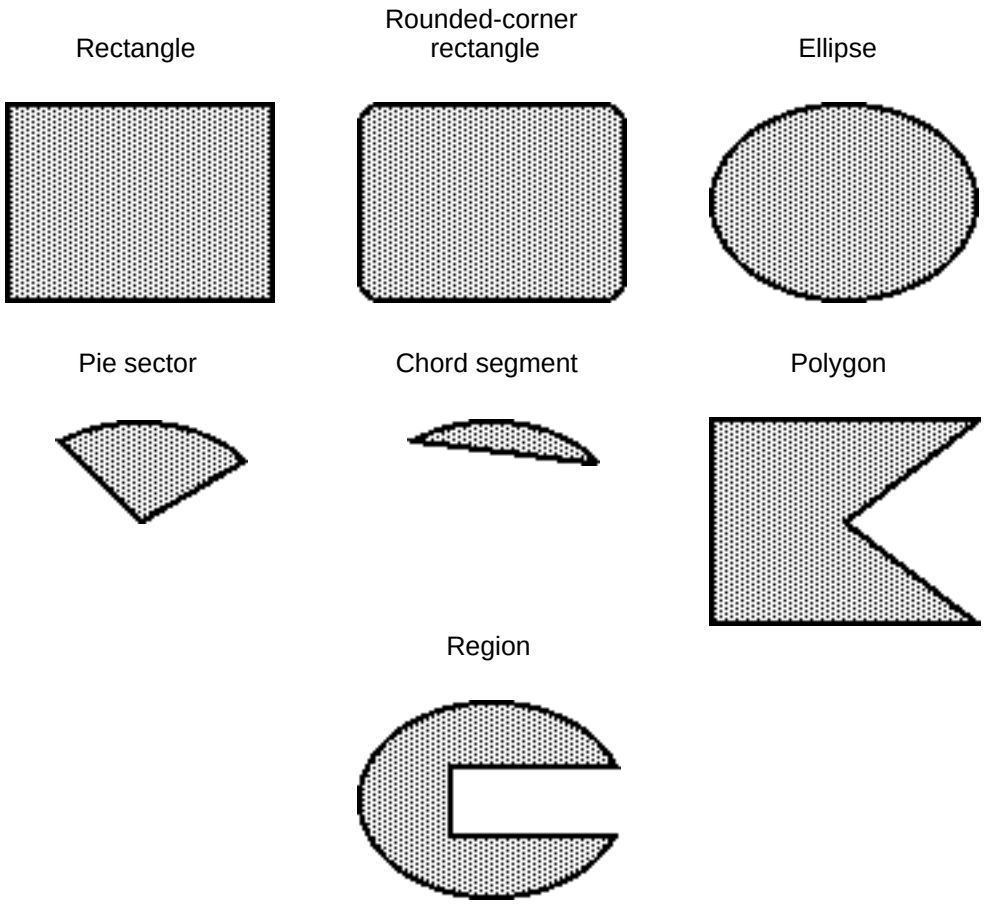


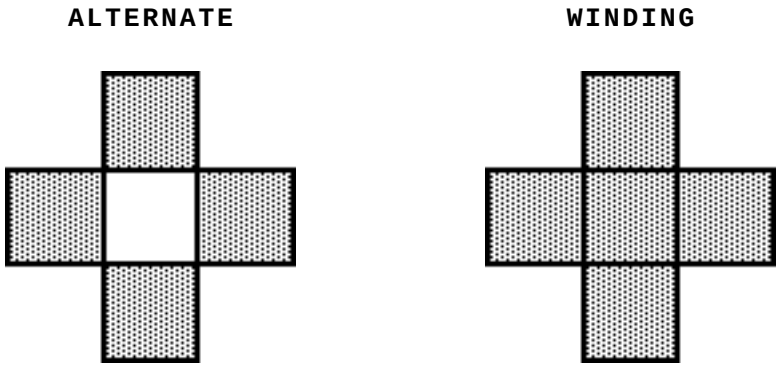
Table 6–13. Shape-drawing functions

Function	Mac counterparts	Purpose
`Rectangle`	`FrameRect` `PaintRect`	Outline and fill rectangle with current pen and brush
`FrameRect`	`FrameRect`	Outline rectangle with specified brush
`FillRect`	`FillRect`	Fill rectangle with specified brush
`InvertRect`	`InvertRect`	Invert rectangle
`RoundRect`	`FrameRoundRect` `PaintRoundRect`	Outline and fill rounded-corner rectangle with current pen and brush
`Ellipse`	`FrameOval` `PaintOval`	Outline and fill ellipse with current pen and brush
`Pie`	`FrameArc` `PaintArc`	Outline and fill elliptical sector with current pen and brush
`Chord`	————	Outline and fill elliptical segment with current pen and brush
`Polygon`	`FramePoly` `PaintPoly`	Outline and fill polygon with current pen and brush
`PolyPolygon`	————	Outline and fill multiple polygons with current pen and brush
`StrokePath`	————	Outline path with current pen
`FillPath`	————	Fill path with current brush
`StrokeAndFillPath`	————	Outline and fill path with current pen and brush
`FrameRgn`	`FrameRgn`	Outline region with specified brush
`PaintRgn`	`PaintRgn`	Fill region with current brush
`FillRgn`	`FillRgn`	Fill region with specified brush
`InvertRgn`	`InvertRgn`	Invert region

the point where it crosses the second. A sector is the figure bounded by the arc and the two radials themselves; a segment is bounded by the arc and a straight line, or *chord*, joining its endpoints (see Figure 6–8).

Each of the shapes has a general drawing function associated with it, as listed in Table 6–13: **`Rectangle`**, **`RoundRect`**, **`Ellipse`**, **`Pie`**, **`Chord`**, **`Polygon`**, and **`PolyPolygon`**. Each of these functions operates in a given device context, using the context's current pen to draw the outline of the specified figure and the current brush to fill its interior. Unlike the Macintosh **`Frame`** operation, which is always confined within the mathematical boundary of the figure, the Windows pen centers on the figure boundary, drawing half its width inside the figure and half outside (unless the pen was created with the style option **`PS_INSIDEFRAME`**, in which case it draws its full width inside the figure boundary, as on the Macintosh). In the general case, the Macintosh **`Erase`** and **`Invert`** operations are not directly available, though you can achieve their effects by filling the figure with the window's background brush (for **`Erase`**) or by using the **`R2_NOT`** drawing mode (for **`Invert`**).

Figure 6–9. Fill modes



For arbitrarily-shaped paths and regions, there are no analogous general drawing functions **`Path`** and **`Region`**. Instead, there are specialized functions for specific drawing operations. For paths, the function **`StrokePath`** draws the outline, **`FillPath`** fills the interior, and **`StrokeAndFillPath`** does both in a single operation. For regions, **`FrameRgn`** draws the outline with a specified brush, **`FillRgn`** fills the interior with a specified brush, **`PaintRgn`** fills the interior with the context's current brush, and **`InvertRgn`** inverts the colors of all existing pixels in the interior. (In a monochrome bitmap, inverting reverses white pixels to black and vice versa; in color, the results are not intuitively predictable.) For convenience, there's also a similar set of specialized functions for rectangles: **`FrameRect`**, **`FillRect`**, and **`InvertRect`**.

A special consideration arises in the case of polygons, paths, or regions that intersect themselves in complex ways (see Figure 6–9). In filling the interior of such a shape with a brush, what constitutes the interior of the shape? There are two different ways of answering this question, depending on an attribute of the device context called the *fill mode*. In **`ALTERNATE`** fill mode, alternating closed areas are considered to lie inside or outside the shape; in **`WINDING`** mode, any area completely encircled at least once by the pen in the course of drawing the shape's outline is considered to lie in the interior. Note, however, that the direction in which the pen draws is significant: if the two rectangles making up the region in the figure were drawn inopposite directions (one clockwise and the other counterclockwise), then the two circuits of the pen would cancel each other out, leaving the central overlapping area outside the boundary instead of inside it. The result would thus end up looking like the first shape in the figure, even when drawn in **`WINDING`** mode. The Windows function **`SetPolyFillMode`** sets a device context's fill mode, and **`GetPolyFillMode`** retrieves it.

Low-Level Pixel Transfer

The most basic drawing functions are those that transfer pixels directly from one bitmap to another (Table 6–14). Of these, the most basic of all is **`BitBlt`** ("bit-level block transfer," pronounced "bit-blit"). This function transfers the graphical contents of a rectangle from one device context into an equal-size rectangle in another device context, under the control of the destination context's current brush. The rectangle may be translated (shifted) horizontally and

vertically to a new location in coordinate space, but no scaling (stretching or shrinking) is performed, because the source and destination rectangles are constrained to be the same size.

A *transfer mode,* supplied as a parameter, defines how the pixels of the source rectangle, brush, and destination rectangle combine to produce the pixels of the result. This transfer mode is a *ternary raster operation* (ROP) resembling the binary ROPs we looked at earlier, but with three operands instead of only two. Three operands can produce $2^3 = 8$ bit combinations, so the total number of possible ternary modes is $2^8 = 256$. Only a few of these are really useful, however; the ones listed in Table 6–15 are the only ones that have names defined in the Win32 interface. For the truly obsessive, all 256 ternary raster operations are listed in an appendix to the *Win32 Programmer's Reference.*

The remaining transfer functions in Table 6–14 allow you to apply various other transformations to the pixels you transfer from the source to the destination bitmap. **`StretchBlt`**, for instance, lets you specify source and destination rectangles of different sizes, stretching or shrinking the results to fit; this makes it more or less equivalent to the Macintosh **`CopyBits`** routine. **`MaskBlt`** accepts a monochrome bit mask as an additional parameter, allowing you to transfer an irregularly-shaped area from one bitmap to another instead of just a rectangle. **`PlgBlt`** maps a source rectangle into a parallelogram in the destination, allowing more sophisticated transformations like rotation and shear in addition to simple translation and scaling. **`PatBlt`** doesn't use a source bitmap at all, but simply fills a rectangle in the destination bitmap with the device context's current brush; Windows uses this function internally to implement the various filling and painting operations that we discussed in the section on shape drawing. Finally, the **`ExtFloodFill`** function (an extended version of an earlier, obsolete function named **`FloodFill`**) accepts the coordinates of a "seed" point and finds the smallest area surrounding

Table 6–14. Bitmap transfer functions

Function	Mac counterpart	Purpose	Transformations
`BitBlt`	————	Copy source rectangle to destination rectangle with current brush	Mode Translation
`StretchBlt`	`CopyBits`	Copy source rectangle to destination rectangle with current brush	Mode Translation Scaling Reflection
`MaskBlt`	`CopyMask`	Mask source rectangle to destination rectangle with current brush	Mode Mask Translation
`PlgBlt`	————	Mask source rectangle to destination parallelogram with current brush	Mask Translation Scaling Reflection Rotation Shear
`PatBlt`	————	Fill destination rectangle with current brush	Mode
`ExtFloodFill`	`SeedFill`	Fill colored area around given point with current brush ("paint can" operation)	————

Table 6–15. Common ternary drawing modes

Mode	Description
SRCCOPY	Copy source to destination
NOTSRCCOPY	Invert source and copy to destination
SRCPAINT	Merge source with destination
MERGEPAINT	Invert source and merge with destination
NOTSRCERASE	Merge source with destination and invert
SRCAND	Mask source with destination
SRCERASE	Mask source with inverse of destination
SRCINVERT	Selectively invert destination with source
PATCOPY	Copy brush to destination
PATPAINT	Invert source and merge with brush and destination
MERGECOPY	Mask source with brush
PATINVERT	Selectively invert destination with brush
DSTINVERT	Invert destination
BLACKNESS	Paint solid black
WHITENESS	Paint solid white

that point that is either filled with or bounded by pixels of a specified color; this makes it useful, like the Macintosh **SeedFill** routine, for implementing the operation of the "paint bucket" tool found in many graphics programs.

Metafiles

The Windows equivalent of a QuickDraw picture is a *metafile.* There are actually two kinds of metafile, an old "Windows format" that's still supported for backward compatibility and a new "enhanced format" that everybody is supposed to use from now on. "From now on" starts now, so we'll confine our discussion to the enhanced format only.

A metafile, like a QuickDraw picture, is a "tape recording" of a sequence of GDI calls that can be "played back" to reconstruct the image. You record the operations by issuing them to a special *metafile device context,* created with the Windows function **CreateEnhMetaFile**. You supply a handle to an existing device context, called the *reference device context,* and get one back to a metafile device context with the same coordinate system, color depth, and so on. (You can leave the reference context unspecified, in which case the metafile context will be made compatible with the current screen.) The **CreateEnhMetaFile** function also accepts a bounding rectangle (like the picture frame on the Macintosh) and a file name in which to record the metafile. If you don't supply a file name, the metafile will be memory-based and will not persist past the end of your program. You can also supply an optional string of descriptive text to be stored in the

Table 6–16. Common metafile functions

Function	Mac counterpart	Purpose
`CreateEnhMetaFile`	`OpenPicture`	Open metafile definition
`GetEnhMetaFile`	`GetPicture`	Open existing metafile
`CopyEnhMetaFile`	————	Copy metafile
`PlayEnhMetaFile`	`DrawPicture`	Draw metafile contents
`CloseEnhMetaFile`	`ClosePicture`	Close metafile definition
`DeleteEnhMetaFile`	`KillPicture`	Delete metafile handle

metafile's header, identifying its contents, the program that created it, and so forth. The function **`GetEnhMetaFileDescription`** retrieves this descriptive string.

Once you have a metafile device context, you can use it in GDI operations just like any other device context. Instead of drawing anything directly, however, your GDI calls will simply be recorded into the accumulating metafile. When you're finished defining the picture, you close it with a call to **`CloseEnhMetaFile`**. This destroys the metafile device context and returns a handle to the completed metafile, of type **`HENHMETAFILE`**. You can then pass this handle to the Windows function **`PlayEnhMetaFile`** to draw the recorded picture into any other device context. (You can even draw it into another metafile context, embedding one picture within another.) Just as on the Macintosh, you supply a target rectangle in which to draw the picture; if the target rectangle's dimensions don't match those of the frame rectangle specified when the metafile was first created, it will be scaled to fit.

Putting all this together, the code for recording and playing back a metafile looks schematically like this:

```
metaContext = CreateEnhMetaFile (refContext, frameRect,...);   // Open metafile
   . . . metaContext . . . ;                                   // Draw into picture
thePicture = CloseEnhMetaFile (metaContext);                   // Close metafile

. . . ;

PlayEnhMetaFile (drawingContext, thePicture, targetRect,...);  // Redraw picture
DeleteEnhMetaFile (thePicture);                                // Delete picture
```

The closing call to **`DeleteEnhMetaFile`** deletes the metafile handle (**`thePicture`** in the example). If the metafile was memory-based, this call destroys it. If it was file-based, the file is closed but not destroyed; it remains available on the disk to be reopened and reused later, even by another program or another session of the same program, with the Windows function **`GetEnhMetaFile`**.

Table 6–16 summarizes the metafile functions we've discussed. All of them treat the metafile as a "black box" that you can use without knowing anything about what's inside. There are additional functions that open up the metafile and directly access or manipulate its contents, but they're beyond the scope of our discussion. Look in the *Win32 Programmer's Reference* if you want the details.

The Same Thing...

- The Macintosh does its drawing with the QuickDraw graphics routines.
- Windows does its drawing with the Graphics Device Interface (GDI).

Graphical Data Structures

- The Macintosh has basic data structures representing points, rectangles, and RGB colors.
- Windows has basic data structures representing points, rectangles, and RGB colors.
- The Macintosh has regions.
- Windows has regions.

Device Contexts

- The Macintosh does all of its drawing in a graphics port.
- Windows does all of its drawing in a device context.
- Macintosh graphics ports can be used for drawing into a window, to the full screen, to a printer, or to an offscreen bitmap.
- Windows device contexts can be used for drawing into a window, to the full screen, to a printer, or to an offscreen bitmap.
- A Macintosh graphics port is a complete drawing environment containing all the information needed to draw lines, shapes, bitmaps, and text.
- A Windows device context is a complete drawing environment containing all the information needed to draw lines, shapes, bitmaps, and text.
- A Macintosh graphics port has a clipping region.
- A Windows device context has a clipping region.

Line Drawing

- A Macintosh graphics port has a pen for drawing lines.
- A Windows device context has a pen for drawing lines and curves.
- The Macintosh graphics pen has a width and height, a color, a pattern, and a drawing mode.
- The Windows graphics pen has a width, a color, a pattern, and a drawing mode.
- On the Macintosh, you draw a line by moving the pen to the starting point and then drawing to the ending point.
- In Windows, you draw a line by moving the pen to the starting point and then drawing to the ending point.
- The Macintosh can draw circular or elliptical arcs.
- Windows can draw circular or elliptical arcs.

Shape Drawing

- The Macintosh can fill an area with a pattern.
- Windows can fill an area with a brush.
- The Macintosh can draw rectangles, rounded-corner rectangles, ovals, wedges, polygons, and regions.
- Windows can draw rectangles, rounded-corner rectangles, ellipses, pie sectors, chord segments, polygons, and regions.
- The Macintosh can frame a shape with the pen, fill it with a pattern, or both.
- Windows can frame a shape with the pen, fill it with a brush, or both.

Low-Level Pixel Transfer

- The Macintosh **`CopyBits`** operation transfers pixels from a rectangle in one bitmap to a rectangle in another.
- The Windows **`BitBlt`** operation transfers pixels from a rectangle in one bitmap to a rectangle in another.
- The Macintosh **`CopyBits`** operation can stretch or shrink pixels to scale between source and destination rectangles of different sizes.
- The Windows **`StretchBlt`** operation can stretch or shrink pixels to scale between source and destination rectangles of different sizes.
- The Macintosh **`CopyMask`** operation copies pixels from one bitmap to another under control of a mask.
- The Windows **`MaskBlt`** operation copies pixels from one bitmap to another under control of a mask.

Metafiles

- The Macintosh can record a series of drawing operations in a picture for later playback.
- Windows can record a series of drawing operations in a metafile for later playback.

...Only Different

Graphical Data Structures

- The coordinates of a Macintosh point are called **`v`** and **`h`**.
- The coordinates of a Windows point use the conventional mathematical names, **`x`** and **`y`**.
- The vertical (**`v`**) coordinate of a Macintosh point precedes the horizontal (**`h`**).
- The horizontal (**`x`**) coordinate of a Windows point precedes the vertical (**`y`**), following standard mathematical convention.

• Macintosh regions can be any arbitrary shape.	• Windows regions are limited to combinations of certain basic shapes: rectangles, rounded-corner rectangles, ellipses, and polygons. Windows *paths* can be any arbitrary shape.
• Macintosh regions are defined by drawing their boundaries with QuickDraw operations.	• Windows regions are built up by combining the basic shapes to form more complex regions. Windows *paths* are defined by drawing their boundaries with GDI operations.

Device Contexts

• The Macintosh maintains a current graphics port in which all drawing takes place.	• Windows has no current device context; all GDI operations take a device context as an explicit parameter.
• A Macintosh graphics port is a public data structure whose attributes you can manipulate directly.	• A Windows device context is a private data structure; Windows provides functions you can call to manipulate its attributes.
• Every Macintosh window has a graphics port permanently embedded within it.	• A Windows window typically is not permanently associated with a device context.
• To draw into a Macintosh window, you use its permanently embedded graphics port.	• To draw into a Windows window, you normally borrow a device context from a common pool and then return it when you're finished.
• A Macintosh graphics port retains its state from one use to the next.	• A Windows device context must normally be reinitialized each time you use it.
• In Macintosh bitmaps, a **`0`** bit represents a white pixel and a **`1`** represents black.	• In Windows bitmaps, a **`0`** bit represents a black pixel and **`1`** represents white.
• The scan lines in a Macintosh bitmap are stored from top to bottom.	• The scan lines in a Windows bitmap are normally stored from bottom to top.

Coordinate Systems

• Coordinates in a Macintosh graphics port are always expressed in physical pixels relative to a given graphics device.	• Coordinates in a Windows device context can be expressed in a variety of device-dependent and device-independent coordinate systems.

- Coordinates in a Macintosh graphics port are always scaled uniformly in the horizontal and vertical directions.
- Coordinates in a Windows device context may be scaled differently in the horizontal and vertical directions.
- Vertical coordinates in a Macintosh graphics port increase from top to bottom.
- Vertical coordinates in a Windows device context can increase from top to bottom or from bottom to top, depending on the mapping mode.
- The coordinate axes in a Macintosh graphics port are always perpendicular and oriented in a straight horizontal and vertical direction.
- The coordinate axes in a Windows device context may be rotated or sheared, depending on the world transform.

Line Drawing

- The Macintosh graphics pen always draws solid lines.
- The Windows graphics pen can draw lines in a variety of solid, dashed, or dotted styles.
- The Macintosh graphics pen has no control over line joins and end caps.
- The Windows graphics pen can produce a variety of line joins and end caps.
- The Macintosh graphics pen can be positioned either in absolute coordinates or by a relative offset from its current position.
- The Windows graphics pen can be positioned only in absolute coordinates.
- The Macintosh graphics pen always draws both the starting and ending points of a line.
- The Windows graphics pen draws the starting point of a line, but not the ending point.
- The Macintosh doesn’t draw irregular curves.
- Windows can draw arbitrary Bézier curves.
- In a single operation, the Macintosh pen can draw a straight line or a circular or elliptical arc.
- In a single operation, the Windows pen can draw a series of one or more connected or unconnected straight lines, circular or elliptical arcs, and Bézier curves.

Shape Drawing

- The Macintosh has no way to control the alignment of a pattern with respect to the coordinate system.
- A Windows device context has a brush alignment that controls the way brushes align with the coordinate system.

- The Macintosh has eight possible drawing modes for combining the pixels of the pen or pattern with those of the target bitmap.
- When framing a shape, the Macintosh pen always draws entirely within the shape's mathematical boundary.
- The Macintosh can erase or invert a shape.

- Windows has sixteen possible drawing modes for combining the pixels of the pen or brush with those of the target bitmap.
- When framing a shape, the Windows pen normally centers on the shape's mathematical boundary, drawing half inside and half outside it.
- In the general case, Windows has no specific operations for erasing or inverting a shape.

Low-Level Pixel Transfer

- The Macintosh **`CopyBits`** operation cannot rotate or shear the contents of a rectangle when copying pixels from one bitmap to another.

- The Windows **`PlgBlt`** operation can rotate or shear the contents of a rectangle when copying pixels from one bitmap to another.

Metafiles

- You record a Macintosh picture in an ordinary graphics port, between calls to **`OpenPicture`** and **`ClosePicture`**.

- You record a Windows metafile in a special metafile device context, between calls to **`CreateEnhMetaFile`** and **`CloseEnhMetaFile`**.

Chapter 7

Controls

Like a Macintosh program, a Windows program can use *controls* to gather information from the user about how to perform its operations. Controls come in various flavors (Figure 7–1), all more or less resembling their Macintosh counterparts:

- *Pushbuttons* trigger an immediate action of some kind.
- *Checkboxes* toggle an option on or off.
- *Radio buttons* offer a choice among two or more mutually exclusive options.
- *Edit controls* allow text entry from the keyboard with cut-and-paste editing.
- *List boxes* offer a scrollable list of items, such as file names.
- *Combo boxes* combine a scrollable list with an editable text box, allowing the user the choice of selecting an item from the list or typing one from the keyboard.
- *Scroll bars* adjust a setting over a continuous range.

Figure 7–1. Standard controls

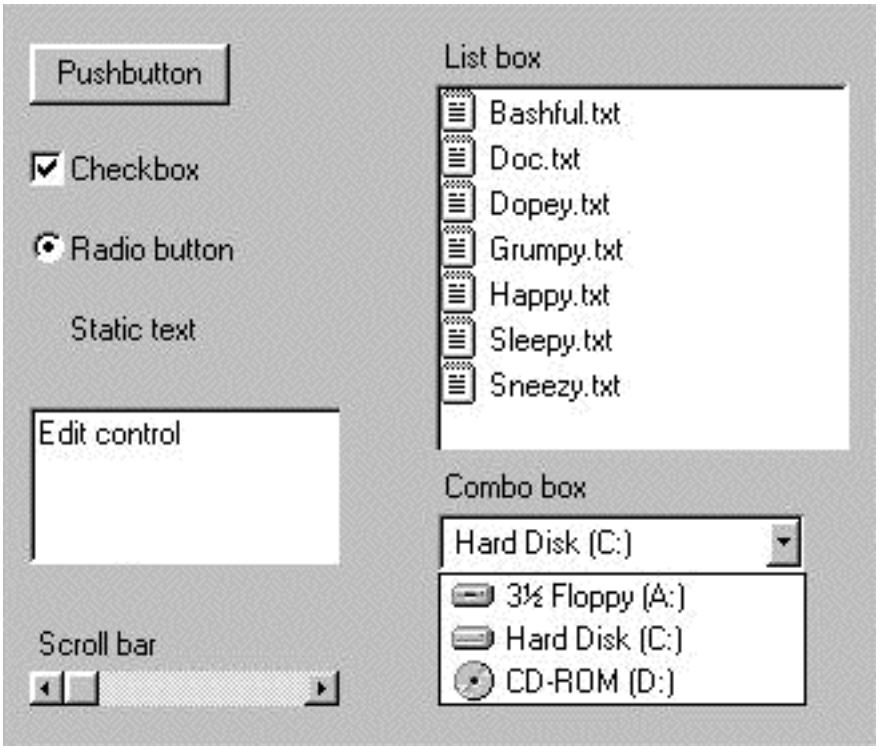


Table 7–1. Predefined control classes

Class name	Control types
BUTTON	Pushbuttons, checkboxes, radio buttons
EDIT	Editable text
LISTBOX	List boxes
COMBOBOX	Combo boxes
SCROLLBAR	Scroll bars
STATIC	Static text, icons, rectangles, frames

- *Static controls* accept no input from the mouse and keyboard, but merely display information to the user on the screen.

The biggest difference between Windows controls and those on the Macintosh is that in Windows, *a control is a window.* It belongs to a window class and has a window procedure of its own. This makes the control much more of an active entity in Windows than it is on the Macintosh: it can gather its own mouse clicks and keystrokes, redraw itself when it's uncovered on the screen, and so forth. Windows provides built-in window classes (Table 7–1) for the standard control types shown in the figure, with predefined window procedures defining their appearance and behavior. We'll be discussing most of these standard controls in this chapter and the next. For most programs, the standard controls are all you'll ever need; you can, of course, define your own custom control classes if your application calls for them.

Because it has a window procedure of its own, a Windows control can take care of much of its own business: it doesn't have to just sit back and wait for the program to analyze the user's actions, determine that they apply to the control, and tell the control what to do. It's true, of course, that Macintosh controls also take some active part in their own operation. Like a window, a Macintosh control has a definition function that responds to messages it receives from the Toolbox. However, again like a window, the set of messages a control can receive is strictly limited (see Table 7–2). Moreover, it receives these messages only as a result of explicit actions

Table 7–2. Macintosh control messages

Message code	Meaning
initCntl	Initialize new control
calcCntlRgn	Calculate region occupied by control
calcThumbRgn	Calculate region occupied by indicator
drawCntl	Draw all or part of control
testCntl	Find part of control where mouse was pressed
thumbCntl	Calculate boundaries for dragging indicator
dragCntl	Drag control or indicator to follow mouse
autoTrack	Perform feedback action while dragging indicator
posCntl	Reposition indicator and update setting
dispCntl	Prepare to dispose of control

by the application program. The Toolbox sends the control a **DrawCntl** message, for instance, only when the program calls a Toolbox routine such as **DrawControls** or **UpdateControls**. Similarly, the control receives a **testCntl** message when the program calls **FindControl** or **TestControl**, and a **dragCntl** message when the program calls **TrackControl** or **DragControl**. It's still up to the program itself to process the user's mouse actions and call these various Toolbox routines at the appropriate times. In Windows, the control takes much more responsibility for its own behavior, gathering the user's input for itself and interacting with the program's main window procedure to implement the required response.

Notification and Control Messages

A Windows control is always a child of some other window, and appears on the screen within the parent window's client area. Typically, the parent window is a dialog box defined by a template resource; you define the control as part of the template, using resource-building tools such as the Visual C++ interactive resource editor. However, a parent window can also create its own controls as part of its response to the **WM_CREATE** message sent by the system when the parent window itself is created. In WiniEdit, for instance, the main document window uses an edit control to handle all of its text editing operations. The edit control is created with the following statements in WiniEdit's **DoCreate** routine, called from the main window procedure in response to the **WM_CREATE** message:

```
editStyle = WS_CHILD |                          // Child window
            WS_VISIBLE |                        //    visible on screen
            WS_VSCROLL |                        //    vertical scroll bar
            ES_AUTOVSCROLL |                    //    vertical autoscroll
            ES_MULTILINE |                      //    multiple lines of text
            ES_LEFT;                            //    flush-left alignment

TheEditor = CreateWindow ("EDIT",               // Standard edit control
                          NULL,                 // No title
                          editStyle,            // Style options as above
                          0, 0, 0, 0,           // Pos. and size will be set later
                          thisWindow,           // Main window is the parent
                          HMENU(Edit_Control),  // Child identifier
                          ThisInstance,         // Current instance is the owner
                          NULL);                // No special creation parameters
```

Notice that the Windows function used to create the control is **CreateWindow**, since the control is simply a window like any other. Its window class is identified by the string **"EDIT"**, the name of the predefined class for edit controls. The style options that specify the control's properties are a combination of ordinary window styles, (prefixed with **WS_**), which we've already discussed in Chapter 4, and others (prefixed with **ES_**, for "edit style") that are specific to edit controls; we'll be looking at these latter in more detail when we talk about text handling in Chapter 8. The control's child window identifier, **Edit_Control**, is a program constant defined in WiniEdit's resource header file, as you can see by referring back a few chapters to Listing 2–2. The remaining parameters to **CreateWindow** are self-explanatory.

Once a control is created, it communicates with its parent window via the normal Windows message mechanism. The control sends *notification messages* to the parent and receives *control messages* from the parent. The possible message types and their meanings vary from one type of

Table 7–3. General control messages

Message type	Meaning
WM_SETTEXT	Set control's text
WM_GETTEXT	Get control's text
WM_GETTEXTLENGTH	Get length of control's text in characters
WM_SETFONT	Set font for displaying control's text
WM_GETFONT	Get font for displaying control's text
WM_SETFOCUS	Acquire input focus
WM_KILLFOCUS	Relinquish input focus

control to another. For instance, a button control (pushbutton, checkbox, or radio button) sends the notification **BN_CLICKED** (**BN_** for "button notification") to its parent window when the user clicks it with the mouse; if the button is a checkbox or radio button, the parent can send it the message **BM_GETCHECK** (**BM_** for "button message") to find out whether the button is currently toggled on (checked) or off (unchecked). A list box sends the notification message **LBN_SELCHANGE** when the selected list item changes, and can receive the control message **LB_GETSEL** to ask it for its currently selected item. These are all unqueued messages, sent directly to the control's or parent's window procedure via the Windows function **SendMessage**.

The messages a control receives from its parent window aren't limited to those defined by the given control type. Because it's actually a window, a control can receive any of the standard **WM_** messages that any other window accepts. Of course, not all such messages are meaningful for controls: the control's window procedure is free to ignore any window message that doesn't apply. There are, however, certain standard window messages that are generally applicable to controls and that most controls should be prepared to accept. For instance, any control that displays text (such as a button label or the contents of an editable text control) should respond to messages such as **WM_SETTEXT**, **WM_GETTEXTLENGTH**, and **WM_SETFONT**. Table 7–3 lists some common messages in this category.

Notification messages from a control to its parent window are a little different. Control-specific notifications such as **BN_CLICKED** for a button or **LBN_SELCHANGE** for a list box aren't actually full-fledged message types in their own right. Instead, all notifications fall into a single message type, **WM_COMMAND**; the specific notification code is passed in the high-order word of the message's **wParam** parameter. The low-order word of this parameter contains the control's child window identifier, an integer constant defined by the application program (such as **Edit_Control** for a WiniEdit window's edit control). The **lParam** parameter holds a handle to the control sending the notification.

> As we'll see in Chapter 9, this same **WM_COMMAND** message is also used to report to the program when the user has chosen a menu command. In that case, the message carries a special notification code to show that it refers to a menu command instead of a control: **0** if the command was chosen with the mouse from an onscreen menu, or **1** if it was invoked by a keyboard shortcut (called an *accelerator*). Note that this means you should never use **0** or **1** as the value of any of your own control identifiers.

Keyboard Interface

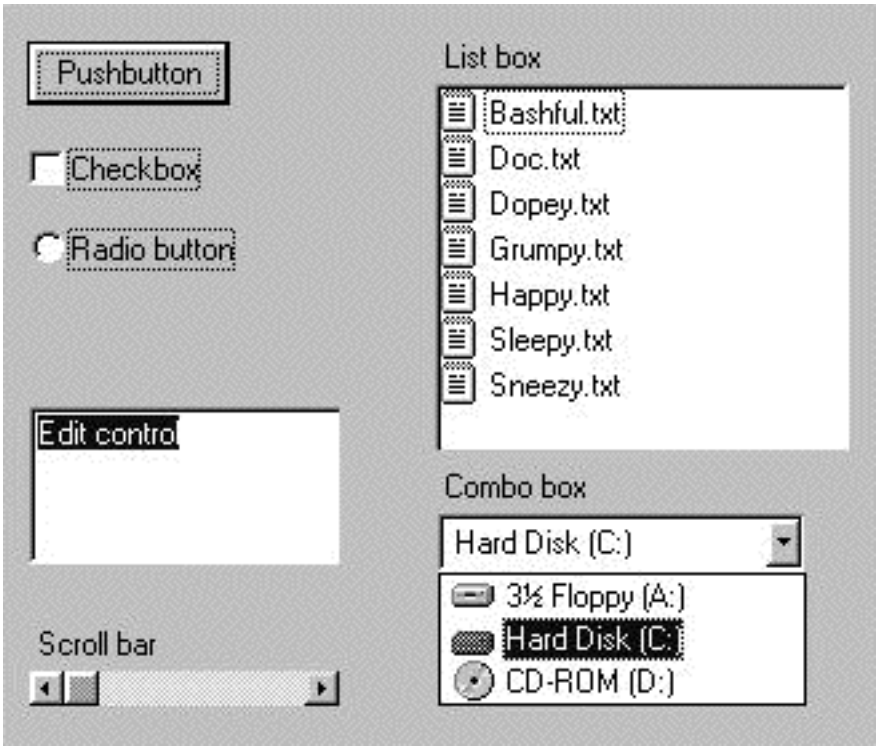
Because not every user has a mouse, Windows provides an alternative, keyboard-based interface for manipulating controls on the screen. In Chapter 5, we learned that all keyboard input is directed to the single window that has the *input focus* at any given time. The focus always belongs to either the active window or one of its children; so in order for a control to respond to the keyboard, its parent window must pass the focus to the control on becoming active. The window can do this by processing the message **`WM_SETFOCUS`** to learn when it (the parent window) has acquired the input focus, and calling the Windows function **`SetFocus`** to relay the focus to one of its controls. WiniEdit's document window, for instance, passes the focus to the edit control that handles its text editing (see Listing 7–1).

Once a control has the input focus, it can process the user's keyboard input in any way it considers appropriate. An edit control, for instance, accepts characters typed from the keyboard and inserts them into its editable text at the insertion caret; a list box moves its selection to the first item in the list beginning with the specified character; a button control responds to the space bar as if it had been clicked with the mouse; and so on. The user can use the Tab and Shift-Tab keys to cycle the focus forward or backward, respectively, from one control or group of controls to the next, and the arrow keys to cycle among controls within a group (such as a group of radio buttons). The control with the focus identifies itself with some form of highlighting on the screen: for instance, an edit control inverts or colors its selection, and a button surrounds its label with a dotted "focus rectangle" (see Figure 7–2). A window can also have a default pushbutton, displayed with a heavy black outline or drop shadow, which is invoked automatically when the user presses the Enter key.

Listing 7–1. Setting the input focus

```
HWND  TheEditor;                                    // Handle to edit control

VOID DoSetFocus (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_SETFOCUS message.

   {
      SetFocus (TheEditor);                         // Pass focus to the edit control

   }  /* end DoSetFocus */
```

Figure 7–2. Highlighted controls



Buttons

On the Macintosh, three related types of control—pushbuttons, checkboxes, and radio buttons—are all implemented by the same standard, system-defined control definition function. From an internal point of view, all three work in essentially the same way: the differences among them are purely cosmetic, affecting the way the appear on the screen but not their underlying behavior. The control definition function distinguishes the three types by means of a *variation code* that you supply in the last four bits of the control definition ID when you create the control. This variation code tells the definition function what format to use when drawing the control on the screen.

In Windows, too, these three control types all belong to the same window class and share the same window procedure. To create a control of any of the three types, you call **`CreateWindow`** and specify the predefined name **`BUTTON`** for the window class. The style option flags in the function's **`dwStyle`** parameter identify which particular type of button you want.

Table 7–4. Button style options

Style name	Meaning
BS_PUSHBUTTON	Standard pushbutton
BS_DEFPUSHBUTTON	Default pushbutton
BS_CHECKBOX	Standard checkbox
BS_AUTOCHECKBOX	Automatic checkbox
BS_3STATE	Three-state checkbox
BS_AUTO3STATE	Automatic three-state checkbox
BS_RADIOBUTTON	Standard radio button
BS_AUTORADIOBUTTON	Automatic radio button
BS_OWNERDRAW	Custom button drawn by owner
BS_GROUPBOX	Box for grouping other controls
BS_TEXT	Displays text
BS_LEFTTEXT	Displays text at left
BS_MONO	All text on one line
BS_BITMAP	Displays a graphical bitmap
BS_ICON	Displays an icon

Table 7–4 lists the available style options pertaining to button controls, all of which begin with the prefix **BS_** for "button style." Most of them are used to identify the specific type of button you want to create, and we'll talk about them further in the sections on individual button types. The remaining options specify the way the button is to be labeled on the screen: with a text label (**BS_TEXT**), a graphical bitmap (**BS_BITMAP**), or an icon (**BS_ICON**). Text labels are, of course, by far the most common. By default, a checkbox or radio button will display its label to the right of the button itself, as in Figure 7–1; the **BS_LEFTTEXT** style alters this behavior and places the label at the left instead. The style **BS_MONO** forces all of the label text to be kept on a single line instead of wrapping, if necessary, to multiple lines.

Button Messages

The only significant notification message that a button control sends to its parent window is **BN_CLICKED**, which reports that the user has clicked the button with the mouse or invoked it from the keyboard with the Enter key or the space bar. (There are a few other button notifications, but they're holdovers from earlier versions of Windows and are obsolete in Windows 95.) The parent window's window procedure can then respond to the **BN_CLICKED** notification by, for instance, executing the action represented by a pushbutton or toggling off the other buttons in a group of radio buttons.

Table 7–5. Button control messages

Message type	Meaning
BM_CLICK	Simulate mouse click
BM_GETCHECK	Is button checked?
BM_SETCHECK	Check or uncheck
BM_GETSTATE	Get full state (check, highlight, focus)
BM_SETSTATE	Set full state (check, highlight, focus)
BM_SETSTYLE	Set style options

In the other direction, in addition to the standard (**WM_**) window messages, a button's parent can send it any of the special-purpose button messages shown in Table 7–5. Probably the most useful of these are **BM_GETCHECK** and **BM_SETCHECK**, which request or change the toggled *check state* of a checkbox or radio button. A similar pair of messages, **BM_GETSTATE** and **BM_SETSTATE**, apply to the button's complete state, which includes not only its check state, but also whether it has the input focus and whether it's currently highlighted on the screen.

Another important button message is **BM_CLICK**, which simulates the effect of the user clicking the button with the mouse or invoking it from the keyboard with the Enter key or the space bar. The message **BM_SETSTYLE** changes a button's style options; this is not something you'd normally want to do, but it might be useful, for instance, for changing a window's default pushbutton in response to changing conditions or circumstances.

Pushbuttons

The BS_PUSHBUTTON style creates a standard pushbutton like the one shown earlier in Figure 7–1. BS_DEFPUSHBUTTON creates a pushbutton with a heavier border, marking it as the default button in its window; pressing the Enter key will invoke this button as if it had been clicked with the mouse. (Naturally, you should avoid creating two buttons with this style in the same window.) If necessary, you can use the button message **BM_SETSTYLE** to change a window's default button:

```
SendMessage (newDefault, BM_SETSTYLE, BS_DEFPUSHBUTTON, TRUE);
```

to mark a new button as the default, and

```
SendMessage (oldDefault, BM_SETSTYLE, BS_PUSHBUTTON, TRUE);
```

to return the previous default button to nondefault status. (The final boolean parameter tells the button to redraw itself on the screen after changing its style.)

Checkboxes

The standard, plain-vanilla checkbox is button style **BS_CHECKBOX**. There's also a three-state version, **BS_3STATE**, that can be placed in a neutral (gray) state, neither checked nor unchecked,

Figure 7–3. Three-state checkboxes

as shown in Figure 7–3. Both kinds send the notification **BN_CLICKED** when clicked with the mouse; the parent window can then respond by toggling or cycling the checkbox's check state:

```
oldState = SendMessage (theCheckbox, BM_GETCHECK, 0, 0);
newState = 1 - oldState;                        // (oldState + 1) % 3 for 3-states
SendMessage (theCheckbox, BM_SETCHECK, newState, 0);
```

(Alternatively, if the checkbox is displayed in a dialog box, you can use the Windows functions **IsDlgButtonChecked** and **CheckDlgButton** to get and set its check state instead of the messages **BM_GETCHECK** and **BM_SETCHECK**.)

These two types of checkbox aren't much different from the ones on the Macintosh: you have to toggle them by hand when the user clicks them with the mouse. It's easier, though, to use the corresponding automatic versions, **BS_AUTOCHECKBOX** or **BS_AUTO3STATE**. Checkboxes created with these styles don't need hand-holding: they look after themselves, changing their own check state when the user clicks them. They still send the **BN_CLICKED** notification to let you know they've been clicked, but you can safely ignore it. All you have to do is ask the checkbox for its current check state when you need the information; the checkbox itself handles the rest.

Radio Buttons

Radio buttons, too, come in both manual and automatic versions, identified at creation time by the style options **BS_RADIOBUTTON** and **BS_AUTORADIOBUTTON**. The difference, of course, is that a radio button is not a self-contained entity like a checkbox, but is connected to a group of other, interrelated radio buttons. Clicking on a checkbox just changes the state of that one control and no others; clicking on a radio button is supposed to turn that button on and, at the same time, turn off all of the other buttons in its group.

For manual radio buttons (**BS_RADIOBUTTON**), it's the parent window's responsibility to keep track of which buttons belong to the group and to turn the right ones on and off when one of them is clicked with the mouse. For instance, your window procedure might keep a global array of handles to all the buttons in the group

```
HWND  TheButtons[nButtons];
```

and then do something like this when you get a **BN_CLICKED** notification from one of them:

```
clickedButton = HWND(lParam);                    // Get handle to clicked button

for ( buttonIndex = 0; buttonIndex < nButtons; buttonIndex++ )
   {
      thisButton = TheButtons[buttonIndex];          // Get button from array
      if ( thisButton == clickedButton )             // Is it the one clicked?
         SendMessage (thisButton, BM_SETCHECK, 1, 0); // Check the clicked button
      else
         SendMessage (thisButton, BM_SETCHECK, 0, 0); // Uncheck all others
   }
```

If (as is usually the case) the controls are part of a dialog box, you can use the Windows function **`CheckRadioButton`** to do the job. This function takes four parameters: a handle to the dialog box, the child window ID of the button to be checked, and the IDs of the first and last buttons belonging to the group. In addition to checking the specified button, it also unchecks all others in the group.

Of course, the easiest thing to do is to use automatic radio buttons (**`BS_AUTORADIOBUTTON`**), which take care of all the housekeeping for themselves. But in order for them to work, you have to identify which buttons belong together as a group. To do this, you have to make sure to create all the buttons in the group consecutively, without any other intervening calls to **`CreateWindow`**. The first control in the group must carry the style option **`WS_GROUP`**; then, after you've finished creating all of the buttons in the group, the next window you create (if any) that *doesn't* belong to the group must also have the **`WS_GROUP`** style. Each control you create with this style begins a new group; the group continues up to, but not including, the next **`WS_GROUP`** control. The **`WS_GROUP`** style thus serves to define the boundaries of a group of radio buttons, so that Windows knows which ones to turn off when one of them is clicked.

When you define a group of related controls, it's conventional to surround them with a *group box* to group them together on the screen and indicate their purpose. The group box is itself a control, created with the style option **`BS_GROUPBOX`**. It is a static control, however, meaning that it doesn't respond to the mouse and keyboard; its sole purpose is to convey visual information to the user.

Owner-Drawn Buttons

If the none of the standard button types meet your needs, you can create a customized button with the style option `BS_OWNERDRAW`. Whenever such a button needs to be drawn on the screen, Windows sends you a **`WM_DRAWITEM`** message instead of drawing the button itself. One of the parameters of this message is the child window ID of the control to be drawn; the other is a pointer to a data structure of type **`DRAWITEMSTRUCT`**, which further specifies the drawing operation to be performed. We won't spend any time discussing the details here, but you can look in the *Win32 Programmer's Reference* if you want to know more.

List Boxes

Although the Macintosh has always had scrollable lists as part of its Standard File dialogs, the original Toolbox had no facilities for programs to build and maintain independent list boxes of

their own. This oversight was eventually remedied with the addition of the List Manager package; but because they were added as an afterthought, lists remained a separate entity with a distinct programming interface of their own, separate from that of the standard Toolbox Control Manager.

In Windows, list boxes are just another kind of control, no different in principle from a pushbutton or a checkbox or a radio button. Like any control, a list box is a child of the window in which it appears (usually a dialog box). It has its own window procedure (defined by the built-in window class **`LISTBOX`**), handles its own mouse clicks and keystrokes, and communicates with its parent window by sending notification messages and receiving control messages.

Creating a List Box

Like most controls, a list box is typically displayed in a dialog box and defined as part of the template resource from which the dialog itself is created. You specify its appearance and properties with the same resource-building tools you use to create the dialog template, whether an interactive resource editor or a text-based resource compiler. When your program creates a dialog from the template, the list box (along with any other controls the template calls for) is created for you automatically.

As always, however, you can also create your list box on the fly, by calling the Windows function **`CreateWindow`** and giving **`"LISTBOX"`** as the name of the window class. You can specify any combination of the style options shown in Table 7–6, as well as the ordinary window style options that we learned about in Chapter 4. For instance, although list boxes are created by default without any visible border or scroll bar, you'll usually want to request these features by using the window style options **`WS_BORDER`** and **`WS_VSCROLL`**. For convenience, these and the other most commonly used style options are combined under a single name, **`LBS_STANDARD`**: besides a border and a scroll bar, this also calls for an alphabetically sorted list of text items that reports to its parent window with notification messages when any of its items are selected or deselected with the mouse. We'll discuss the remaining options in the table as they come up.

Table 7–6. List box style options

Style name	Meaning
LBS_STANDARD	Standard list box: border, vertical scroll bar, text items, alphabetical sort, parent notified when items clicked
LBS_MULTIPLESEL	Allow multiple items to be selected
LBS_EXTENDEDSEL	Allow extended selection with Shift key
LBS_HASSTRINGS	Item contents are text strings
LBS_NODATA	Don't allocate space for item data
LBS_SORT	Sort items alphabetically
LBS_WANTKEYBOARDINPUT	Accept keyboard input messages
LBS_NOTIFY	Notify parent window when items are clicked
LBS_MULTICOLUMN	More than one column
LBS_USETABSTOPS	Allow tab characters in item strings
LBS_NOINTEGRALHEIGHT	Allow fractional items to be visible
LBS_DISABLENOSCROLL	Disable scroll bar instead of hiding it when entire list is visible
LBS_NOREDRAW	Don't redraw automatically after changes
LBS_OWNERDRAWFIXED	Custom list box drawn by owner, with fixed item heights
LBS_OWNERDRAWVARIABLE	Custom list box drawn by owner, with variable item heights

Adding and Retrieving List Items

Once you've created a list box, you can manage its contents by sending it the control messages shown in Table 7–7. The great majority of list boxes are created with the style option **LBS_HASSTRINGS**, meaning that each individual list item is represented by a string of text characters. You add items to such a list with the messages **LB_ADDSTRING** and **LB_INSERTSTRING**, both of which take a string pointer as one of their parameters. If the list box also has style **LBS_SORT**, the **LB_ADDSTRING** message automatically places each new string in its proper alphabetical place in the list; if the list is unsorted, new strings are simply appended at the end. **LB_INSERTSTRING** places a new string at a designated position within the list, regardless of the **LBS_SORT** property. (That is, the string is always inserted at the position you specify, even if not in proper alphabetical order.)

One very common use for list boxes, of course, is to present a list of file names for the user to choose from in an input or output operation. The message **LB_ADDFILE** adds a file name to the list, while **LB_DIR** adds the entire contents of a designated directory. In practice, though, you'll rarely need to use these messages to build a file list. For most purposes, it's more convenient to use the standard Open and Save As dialogs from the Common Dialog Box Library, the Windows counterpart to the Macintosh Standard File Package.

After you've added some items to a list box, you can use the message **LB_GETCOUNT** to find out how many items it currently contains. The messages **LB_GETTEXTLENGTH** and **LB_GETTEXT**

Table 7–7. List box control messages for adding and retrieving items

Message type	Meaning
LB_ADDSTRING	Add text item to list
LB_INSERTSTRING	Insert text item at specified position in list
LB_DELETESTRING	Delete text item from list
LB_ADDFILE	Add file name to list
LB_DIR	Add directory contents to list
LB_RESETCONTENT	Remove all items from list
LB_GETCOUNT	Get number of items in list
LB_SETCOUNT	Set number of items in list
LB_GETTEXT	Get text of item
LB_GETTEXTLENGTH	Get length of item's text
LB_GETITEMDATA	Get data value of item
LB_SETITEMDATA	Set data value of item
LB_FINDSTRING	Find item beginning with a given string
LB_FINDSTRINGEXACT	Find item matching a given string

retrieve the length and contents, respectively, of the string at a given indexed position in the list. (All item indices are zero-based: that is, the first item in the list has index number **0**.) You can find an item matching a given string with **LB_FINDSTRING** or **LB_FINDSTRINGEXACT**, delete an item from the list with **LB_DELETESTRING**, or clear the entire list to empty with **LB_RESETCONTENT**.

Scrolling and Display

For list boxes with the normal **LBS_HASSTRINGS** style, the built-in **LISTBOX** window procedure will take care of displaying the list on the screen for you. It normally uses the default system font for the text of each item, but you can change this, if you wish, by sending it the window message **WM_SETFONT**. The standard window procedure also handles all visual feedback to the user, such as highlighting selected items, displaying a focus rectangle when needed, and so on.

Even if you're absolutely certain that the number of items in the list can't exceed the height of the list box, there's no harm in using style option **WS_VSCROLL** to create a vertical scroll bar for scrolling additional items into view: Windows will hide the scroll bar if the list is short enough to be displayed in its entirety. (Style **LBS_DISABLENOSCROLL** overrides this behavior, causing the scroll bar to remain visible on the screen but simply become disabled when no scrolling is necessary.) You can also specify a horizontal scroll bar (style **WS_HSCROLL**), allowing the list to be wider than the list box itself. The messages **LB_SETHORIZONTALEXTENT** and **LB_GETHORIZONTALEXTENT** (see Table 7–8) control the maximum width allowed for the list; the horizontal scroll bar will be hidden if you set the horizontal extent of the list less than the width of the list box.

Table 7–8. List box control messages for scrolling and display

Message type	Meaning
LB_GETITEMRECT	Get enclosing rectangle of item in client coordinates
LB_GETITEMHEIGHT	Get height of item in pixels
LB_SETITEMHEIGHT	Set height of item in pixels
LB_GETTOPINDEX	Get item scrolled to top of list
LB_SETTOPINDEX	Scroll item to top of list
LB_GETHORIZONTALEXTENT	Get scrollable width of list in pixels
LB_SETHORIZONTALEXTENT	Set scrollable width of list in pixels
LB_SETCOLUMNWIDTH	Set column width in pixels
LB_SETTABSTOPS	Set tab stops for list box

When the scroll bar is visible and enabled, Windows handles all of its interactions with the mouse and scrolls the list automatically in response. The message **LB_GETTOPINDEX** returns the index of the first item currently visible in the box; with **LB_SETTOPINDEX**, you can scroll a specified item to the top. If you like, you can give your list a tabular layout by using either the style option **LBS_MULTICOLUMN**, to create a grid of rows and columns, or **LBS_USETABSTOPS**, to allow tab characters to be embedded within the text of individual list items. In the former case, the message **LB_SETCOLUMNWIDTH** sets the width of each column; in the latter, **LB_SETTABSTOPS** sets the horizontal positions of the tab stops.

If necessary, you can find out the coordinates and dimensions of individual items with the messages **LB_GETITEMRECT** and **LB_GETITEMHEIGHT**. You can specify the item height with **LB_SETITEMHEIGHT**, but normally you'll just want to let Windows set it by default. If the height of the list box is not exactly equal to an integral number of item heights, any fractional item remaining at the bottom of the box will be suppressed unless the box has style **LBS_NOINTEGRALHEIGHT**.

Owner-Drawn List Boxes

If you don't specify style **LBS_HASSTRINGS** when creating a list box, its items will consist of some other type of data instead of strings. In this case, it's entirely up to your program to define and interpret the values of the list items; Windows takes no direct responsibility for drawing them on the screen or manipulating them in any way. Such a list must be created with either the style option **LBS_OWNERDRAWFIXED** (if all of its items are the same height on the screen) or **LBS_OWNERDRAWVARIABLE** (if the height can vary from one item to another).

You add items to such an owner-drawn list with the same messages, you would use for a list of text items, **LB_ADDSTRING** and **LB_INSERTSTRING**, but Windows treats the parameters to these messages as uninterpreted 32-bit data values rather than as string pointers. When you need to retrieve the data value of an item, you use the message **LB_GETITEMDATA** instead of **LB_GETTEXT**; you can change the value of an existing item with **LB_SETITEMDATA**. Another

alternative is to specify the style option **LBS_NODATA** when creating the list box. In this case, Windows will not allocate space for associating any data at all with each list item; you assume the entire responsibility for keeping track of the item data yourself, using private data structures of your own.

Since Windows doesn't know what the items of an owner-drawn list box represent, it can't draw them on the screen without your help. When drawing the list, it will ask your window procedure to display each item by sending it the message **WM_DRAWITEM**, with a pointer to a descriptor structure (**DRAWITEMSTRUCT**) containing further information about the item to be drawn. Similarly, if the list box is sorted (style **LBS_SORT**), Windows will ask you for help in sorting the items by repeatedly sending you the message **WM_COMPAREITEM** with a pointer to a structure describing a pair of items to be compared. Another message, **WM_MEASUREITEM**, asks you to supply the dimensions of a list item. For owner-drawn list boxes with variable item height (style **LBS_OWNERDRAWVARIABLE**), this message is sent separately for each item; for those with fixed height (**LBS_OWNERDRAWFIXED**), it's sent just once, when the list box is drawn for the first time. See the *Win32 Programmer's Reference* if you need more information on all of these operations.

Selection Management

Depending on the style options you specify, a list box can allow one of three different kinds of selection:

- A *single-selection list box* allows only one of its items to be selected at a time. Clicking an item selects that item and automatically deselects the previous one, if any. This is the most common type of list box and is created by default if neither of the other two options is explicitly specified.
- A *multiple-selection list box* (**LBS_MULTIPLESEL**) allows the user to select any combination of items, even discontiguous ones. Clicking any item toggles its state, from deselected to selected or vice versa, independently of any other item in the list.
- An *extended-selection list box* (**LBS_EXTENDEDSEL**) also allows multiple selections. An unmodified mouse click selects a single item and deselects any others that were previously selected. The selected item becomes the *anchor* for a new selection. Clicking another item while holding down the Shift key extends the selection from the anchor up to and including the item that was clicked. Clicking with the Ctrl key toggles individual items between selected and deselected, as in a multiple-selection list.

The style **LBS_WANTKEYBOARDINPUT** allows the keyboard to be used for selecting list items on systems without a mouse. The arrow keys move the input focus (indicated by a dotted rectangle) from one item to another; the Enter key or space bar acts like a mouse click. In an extended-selection list box, for instance, the keystroke Shift-Enter is equivalent to a Shift-click, extending the selection from the anchor item to the one with the current focus. Typing a character (or a rapid sequence of characters) from the keyboard moves the selection to the first item beginning with that character or characters.

Table 7–9. List box notifications

Notification code	Meaning
LBN_SELCHANGE	Selection changed
LBN_SELCANCEL	Selection canceled
LBN_DBLCLK	Item double-clicked
LBN_SETFOCUS	Input focus acquired
LBN_KILLFOCUS	Input focus relinquished

The standard window procedure for class **LISTBOX** handles all of the user's mouse and keyboard actions and updates the current selection accordingly: highlighting and unhighlighting items as they're selected and deselected, displaying a focus rectangle around the item with the input focus, and so on. Assuming that the list box has style **LBS_NOTIFY** (part of the standard combination of styles denoted by the name **LBS_STANDARD**), it will keep its parent window informed of all this by sending it the notification messages shown in Table 7–9. There's normally no need to process most of these messages, though you might use **LBN_SELCHANGE**, for instance, to update the contents of a second list box that depend on the item selected in the first.

Ordinarily, you don't need to intervene until the user confirms a selection and dismisses the dialog, which you will learn by receiving either an **LBN_DBLCLK** notification from the list box or a **BN_CLICKED** from the dialog's OK button or its equivalent. At that point, you can ask the list box which item or items were selected, using the messages shown in Table 7–10, and act on them accordingly. In a single-selection list box, the messages **LB_GETCURSEL** and **LB_SETCURSEL** retrieve and change the index of the selected item. You can also use **LB_SELECTSTRING** to do a content-based selection, finding and selecting the first item beginning with a given text string. The **LISTBOX** window procedure sends this message to itself to find the item to select when the user types an initial character or prefix string from the keyboard.

In a mutliple-selection list box, the message **LB_GETSELCOUNT** returns the number of items currently selected, and **LB_GETSELITEMS** fills an array with their indices. You can select or deselect an individual item with **LB_SETSEL**, or a range of consecutive items with **LB_SELITEMRANGE**. In an extended-selection list box, the messages **LB_GETANCHORINDEX** and **LB_SETANCHORINDEX** operate on one end of the selection range, **LB_GETCARETINDEX** and **LB_SETCARETINDEX** on the other. All of these messages but the last two apply only to list boxes with the **LBS_MULTIPLESEL** or **LBS_EXTENDEDSEL** style options; they're invalid for single-selection list boxes.

Table 7–10. List box control messages for selection management

Message type	Meaning
LB_GETCURSEL	Get current selection (single-selection list box)
LB_SETCURSEL	Set or clear current selection (single-selection list box)
LB_GETSELCOUNT	Get number of selected items (multiple-selection list box)
LB_GETSELITEMS	Get array of selected item indices (multiple-selection list box)
LB_GETSEL	Get selection state of item
LB_SETSEL	Select or deselect item (multiple-selection list box)
LB_SELITEMRANGE	Select or deselect range of items (multiple- or extended-selection list box)
LB_SELECTSTRING	Select first item beginning with a given string
LB_GETANCHORINDEX	Get selection anchor (extended-selection list box)
LB_SETANCHORINDEX	Set selection anchor (extended-selection list box)
LB_GETCARETINDEX	Get item with input focus
LB_SETCARETINDEX	Set item with input focus

Scroll Bars

From the user's point of view, Windows scroll bars look and work pretty much the same as those on the Macintosh. They have up and down arrows for scrolling the contents of a window one unit at a time (for instance, one line at a time in a window displaying text), a shaft containing a scroll box for moving quickly to a specific place in the document, and areas above and below the scroll box that scroll up or down by a full windowful at a time. Horizontal scroll bars work the same way, with "left" and "right" substituted for "up" and "down." All of this, of course, is perfectly familiar from the Macintosh. One minor difference is that in Windows, the size of the scroll box changes to reflect the portion of the document currently visible in the window; the Macintosh scroll box has a fixed size, indicating the relative position but not the extent of the document's visible portion.

Under the hood, the differences between the two platforms are more apparent. Windows scroll bars are more closely coupled to the structure of a window than they are on the Macintosh. A Macintosh scroll bar is part of its window's content region, not part of the frame. Instead of drawing the scroll bar for you automatically, as it does the title bar or close box, the Toolbox leaves it to you to draw it yourself when you receive an update event for the window's content region. When you receive a mouse-down event in the scroll bar, the Toolbox routine **`FindWindow`** doesn't tell you so: it simply reports that the click was in the content region. You have to issue a further call to **`FindControl`** to learn that the click was in the scroll bar, and in which part of the scroll bar, so you can respond accordingly.

You can handle your scroll bars the analogous way in Windows, too, making them child windows of class **`"SCROLLBAR"`** displayed within the client area of their parent window. Since such a *scroll bar control* is a window in its own right, you create it in the usual way, with

Table 7–11. Scroll bar style options

Style name	Meaning
SBS_VERT	Vertical scroll bar
SBS_RIGHTALIGN	Align on right edge (vertical scroll bar)
SBS_LEFTALIGN	Align on left edge (vertical scroll bar)
SBS_HORZ	Horizontal scroll bar
SBS_BOTTOMALIGN	Align on bottom edge (horizontal scroll bar)
SBS_TOPALIGN	Align on top edge (horizontal scroll bar)

CreateWindow. Table 7–11 shows the style options you can use to specify the scroll bar's properties. The **SBS_VERT** and **SBS_HORZ** options call for a vertically or horizontally oriented scroll bar, respectively. As always in calling **CreateWindow**, you have to supply the coordinates of the window's (in this case, the scroll bar's) rectangle. The alignment options (**SBS_RIGHTALIGN** or **SBS_LEFTALIGN** for vertical scroll bars, **SBS_BOTTOMALIGN** or **SBS_TOPALIGN** for horizontal ones) align the scroll bar along one edge of the specified rectangle but ignore the opposite edge, using the system's standard scroll-bar width or height instead. (You can find out these standard dimensions by calling **GetSystemMetrics** with the selectors shown in Table 7–12.) If you don't specify any of the alignment-related style options, the scroll bar will fill the entire rectangle, without reference to the standard system dimensions.

Usually, though, it's more convenient to make your scroll bars part of the structure of the window itself by specifying the style options **WS_VSCROLL** and **WS_HSCROLL** at the time you create the window. Strictly speaking, such *standard scroll bars* are not controls, but rather form part of the window's nonclient area. Mouse clicks affecting them are hit-tested by the standard **DefWindowProc** window procedure, which then reports them to you in the form of *scroll messages* of type **WM_VSCROLL** or **WM_HSCROLL**. One of the parameters to these messages is a *scroll code* (Table 7–13) identifying the specific event that occurred. For instance, the scroll codes **SB_LINEUP** and **SB_LINEDOWN** represent mouse clicks in the up and down arrows of a vertical scroll bar; **SB_PAGEUP** and **SB_PAGEDOWN** denote clicks in the shaft above and below the scroll box; and so on. If the user presses and holds down the mouse button in one of these

Table 7–12. System metric selectors for scroll bars

Selector	Meaning
SM_CXVSCROLL	Width of scroll arrows (vertical scroll bar)
SM_CYVSCROLL	Height of scroll arrows (vertical scroll bar)
SM_CYVTHUMB	Height of scroll box (vertical scroll bar)
SM_CXVSCROLL	Width of scroll arrows (horizontal scroll bar)
SM_CYVSCROLL	Height of scroll arrows (horizontal scroll bar)
SM_CYVTHUMB	Height of scroll box (horizontal scroll bar)

Table 7–13. Scroll codes

Scroll code	Meaning
SB_LINEUP	Scroll up one unit (vertical scroll bar)
SB_LINEDOWN	Scroll down one unit (vertical scroll bar)
SB_PAGEUP	Scroll up one windowful (vertical scroll bar)
SB_PAGEDOWN	Scroll down one windowful (vertical scroll bar)
SB_LINELEFT	Scroll left one unit (horizontal scroll bar)
SB_LINERIGHT	Scroll right one unit (horizontal scroll bar)
SB_PAGELEFT	Scroll left one windowful (horizontal scroll bar)
SB_PAGERIGHT	Scroll right one windowful (horizontal scroll bar)
SB_TOP	Scroll to top (vertical scroll bar) or extreme left (horizontal scroll bar)
SB_BOTTOM	Scroll to bottom (vertical scroll bar) or extreme right (horizontal scroll bar)
SB_THUMBTRACK	Dragging scroll box
SB_THUMBPOSITION	Scroll box released after drag
SB_ENDSCROLL	Mouse button released after scrolling operation

areas, you will receive a stream of equivalent scroll messages, causing the window's contents to scroll continuously until the button is released.

Your program's window procedure should respond to such scroll messages by examining the scroll code to determine in which direction to scroll and how far, depending on the nature of the information displayed in the window. For instance, a window containing text would respond to an **SB_LINEUP** or **SB_LINEDOWN** scroll code by scrolling up or down by one line height, and to **SB_LINELEFT** or **SB_LINERIGHT** by scrolling some appropriate distance, perhaps an average character width., left or right. Scroll codes such as **SB_PAGEDOWN** or **SB_PAGERIGHT** would scroll in the indicated direction by the full width or height of the window. The Windows function for performing the scrolling is **ScrollWindow**, which shifts the contents of all or part of a window's client area by a specified amount and direction. Information scrolled out at one end is clipped automatically; the area vacated at the other end is accumulated into the window's update region, generating a **WM_PAINT** message (the equivalent of a Macintosh update event) to draw in the new contents of the area.

As on the Macintosh, the current position of the scroll box within a scroll bar's shaft is represented by an integer *scroll position,* which varies within a specified minimum and maximum *scrolling range.* For a standard scroll bar (one created as part of the window's structure with the style option **WS_VSCROLL** or **WS_HSCROLL**), the scrolling range is initially set to run from **0** to **100**; for a scroll bar control, the minimum and maximum are both initialized to **0**. In either case, you can then use the Windows function **SetScrollRange** to adjust the scroll bar to a more convenient range of values, such as the number of lines in a text document. The function **GetScrollRange** returns the minimum and maximum values of a scroll bar's range; **GetScrollPos** and **SetScrollPos** retrieve and change the value of the current scroll

Table 7–14. Common scroll bar functions

Function	Mac counterpart	Purpose
`ShowScrollBar`	`ShowControl`, `HideControl`	Make scroll bar visible or invisible
`EnableScrollBar`	————	Enable or disable scroll arrows
`GetScrollPos`	`GetCtlValue`	Get current scroll position
`SetScrollPos`	`SetCtlValue`	Set scroll position
`GetScrollRange`	`GetCtlMin`, `GetCtlMax`	Get current scrolling range
`SetScrollRange`	`SetCtlMin`, `SetCtlMax`	Set scrolling range
`ScrollWindow`	`ScrollRect`	Scroll contents of window

position within the designated range. After scrolling the window's contents in response to a scroll message, it's up to you to adjust the position of the scroll box with **`SetScrollPos`** to reflect the change. Windows will automatically interpolate the new value and redraw the scroll box in its correct proportional position, depending on the scroll bar's physical dimensions on the screen. You must also remember to adjust the scrolling range when the size of the underlying document changes (for instance, if the number of lines of text changes as a result of the user's typing or editing). If you use an edit control to manage your text editing, it will automatically take care of all the scroll bar adjustments for you, as we'll see in the next chapter.

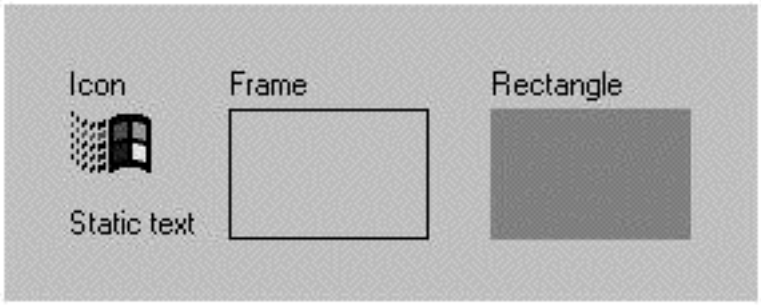
When the user drags the scroll box with the mouse, Windows will send you a series of scroll messages with the scroll code **`SB_THUMBTRACK`**, giving continuous updates on the position of the scroll box as it is being dragged; when the mouse button is released, you will receive a message with the scroll code **`SB_THUMBPOSITION`**, giving the final adjusted scroll position. If you can process the **`SB_THUMBTRACK`** messages fast enough, you can use them to provide continuous scrolling in real time, while the scroll box is being dragged; if your scrolling operations are too elaborate or time-consuming for this kind of real-time response, you can ignore the **`SB_THUMBTRACK`** messages and just reposition the document at the end of the drag, when you receive the final **`SB_THUMBPOSITION`** message. There's also a concluding scroll message, **`SB_ENDSCROLL`**, sent when the button is released at the end of *any* scrolling operation, whether in the scroll bar's arrows, shaft, or scroll box. Typically, you'll just ignore this message, but you may sometimes find it useful for some sort of final terminating action at the conclusion of a scrolling operation.

Table 7–14 summarizes the most common Win32 functions related to scroll bars, with their Macintosh equivalents. We've already touched on most of these functions in our discussion, but see the *Win32 Programmer's Reference* for complete details.

Static Controls

The built-in window class **`STATIC`** defines a category of controls that don't respond to the mouse and keyboard. Although such *static controls* can be included in an ordinary overlapped window,

Figure 7–4. Static controls



they are really designed to be used in dialog boxes, where they display text or graphics to convey information to the user without receiving any communication back. They thus correspond to Macintosh dialog items of type **`StatText`**, **`IconItem`**, and **`PicItem`**.

Static controls are of four main types (Figure 7–4): text, graphics, rectangles, and frames. Each of these, in turn, can appear in a variety of styles. Because they're normally part of a dialog box, you usually create them with an interactive resource editor and load them as part of the dialog template, rather than creating them directly with **`CreateWindow`**. Table 7–15 lists the possible style options, just for the sake of understanding.

Frame and rectangle controls can carry any of three styles each, nominally designated as black, white, and gray. Actually, these represent three of the standard system colors that se learned about in Chapter 6: black refers to the window frame color (**`COLOR_WINDOWFRAME`**), white to the window background color (**`COLOR_WINDOW`**), and gray to the desktop background color (**`COLOR_BACKGROUND`**). These elements are set by default to black, white, and gray, respectively, but the user is free to change these settings with the Display control panel. A static frame control appears as a rectangle outlined in the indicated color; a static rectangle is completely filled with the color.

A static graphical control normally displays an icon (style **`SS_ICON`**), but it can be an arbitrary bitmap (**`SS_BITMAP`**) or a GDI metafile (**`SS_METAPICT`**) instead. For icon controls, you can set the icon displayed by sending the control the message **`STM_SETICON`**, or request a handle to the current icon with **`STM_GETICON`**. For bitmap or metafile controls, the control's text (set with the function **`SetWindowText`** or the message **`WM_SETTEXT`**) gives the resource name of the bitmap or the file name of the metafile to be displayed. The control will automatically size itself to the bounding rectangle of its graphical contents. The styles **`SS_CENTERIMAGE`** and **`SS_RIGHTIMAGE`** anchor the control at its center or bottom-right point, respectively; if neither is specified, it is anchored at the top-left. If the control has to be resized to accommodate a new bitmap, the designated point remains fixed in place while the dimensions of the control are adjusted relative to it.

Table 7–15. Static control style options

Style name	Meaning
SS_BLACKFRAME	Framed box in window frame color
SS_WHITEFRAME	Framed box in window background color
SS_GRAYFRAME	Framed box in screen background color
SS_BLACKRECT	Solid rectangle in window frame color
SS_WHITERECT	Solid rectangle in window background color
SS_GRAYRECT	Solid rectangle in screen background color
SS_BITMAP	Graphical bitmap
SS_ICON	Graphical icon
SS_METAPICT	Graphical metafile (picture)
SS_CENTERIMAGE	Bitmap or icon anchored at center
SS_RIGHTIMAGE	Bitmap or icon anchored at bottom-right corner
SS_SIMPLE	Left-aligned text without tabs or word wrap
SS_LEFT	Left-aligned text with tabs and word wrap
SS_LEFTNOWORDWRAP	Left-aligned text with tabs but no word wrap
SS_CENTER	Centered text with tabs and word wrap
SS_RIGHT	Right-aligned text with tabs and word wrap
SS_NOPREFIX	Literal text, no accelerator interpretation
SS_OWNERDRAW	Owner-drawn item

The simplest form of static text control, **SS_SIMPLE**, just displays a single line of left-aligned text. You specify the text with the **SetWindowText** function or by sending the control the message **WM_SETTEXT**. If the text extends past the right edge of the control, it is simply clipped rather than word-wrapped. If you want word wrap, you have to use one of the styles **SS_LEFT**, **SS_CENTER**, or **SS_RIGHT**. The **SS_LEFTNOWORDWRAP** style clips instead of wrapping, but differs from **SS_SIMPLE** in that it can accommodate multiple lines of text, delimited by carriage-return characters. All of these styles interpret an ampersand character (**&**) embedded within the text as a marker defining the following character as an accelerator, or keyboard shortcut. When displaying the text, the ampersand is omitted and the character following it is underlined to mark it as an accelerator. The style **SS_NOPREFIX**, which can be combined with any of the other styles, suppresses this behavior and causes any ampersands to be displayed literally.

The Same Thing...

- The Macintosh uses controls to gather information from the user about how to perform its operations.
- A Macintosh control is always owned by some window.
- On the Macintosh, controls are usually displayed within a dialog box.
- Macintosh controls include pushbuttons, checkboxes, radio buttons, and scroll bars.
- A Macintosh control has a control definition function.
- The Macintosh implements pushbuttons, checkboxes, and radio buttons all with the same definition function.
- A Macintosh dialog box can have a default pushbutton that’s invoked automatically by the Return or Enter key.
- The Macintosh has a Standard File Package containing standard dialogs for opening and saving files.
- Macintosh scroll bars have up and down (or left and right) arrows for scrolling a line at a time, a shaft for scrolling a windowful at a time, and a scroll box for moving quickly to a specific scroll position.
- Macintosh scroll bars have an integer scroll position bounded by a minimum and maximum range.

- Windows uses controls to gather information from the user about how to perform its operations.
- A Windows control is always a child of some parent window.
- In Windows, controls are usually displayed within a dialog box.
- Windows controls include pushbuttons, checkboxes, radio buttons, and scroll bars.
- A Windows control has a window procedure.
- Windows implements pushbuttons, checkboxes, and radio buttons all with the same window procedure.
- A Windows dialog box can have a default pushbutton that’s invoked automatically by the Enter key.
- Windows has a Common Dialog Box Library containing standard dialogs for opening and saving files.
- Windows scroll bars have up and down (or left and right) arrows for scrolling a line at a time, a shaft for scrolling a windowful at a time, and a scroll box for moving quickly to a specific scroll position.
- Windows scroll bars have an integer scroll position bounded by a minimum and maximum range.

- A Macintosh dialog box can contain static text and graphical items for conveying visual information to the user without receiving any response back.

- A Windows dialog box can contain static text and graphical controls for conveying visual information to the user without receiving any response back.

...Only Different

- The only types of control on the Macintosh are pushbuttons, checkboxes, radio buttons, and scroll bars.

- Windows has several additional types of control, including edit controls, list boxes, combo boxes, and static controls.

- On the Macintosh, a control is a separate type of entity displayed within a window.

- In Windows, a control is a child window displayed within its parent window.

- On the Macintosh, a control is a passive entity that must be manipulated by the program.

- In Windows, a control is an active entity that can send, receive, and process messages in its own right.

- Macintosh controls don’t normally respond to input from the keyboard.

- Windows controls can acquire the input focus and respond to keyboard input.

- Macintosh checkboxes have two states: checked and unchecked.

- Windows checkboxes can have either two or three states: checked, unchecked, and neutral.

- A Macintosh program must explicitly change the state of a checkbox or radio button when the user clicks it with the mouse.

- A Windows program can create automatic checkboxes or radio buttons that change their own state when clicked with the mouse.

- On the Macintosh, scrollable lists are a separate entity, different from controls.

- In Windows, scrollable lists are a type of control.

- The scroll box of a Macintosh scroll bar is always the same fixed size.

- The scroll box of a Windows scroll bar adjusts in size to reflect the portion of a document currently visible in the window.

- A Macintosh scroll bar is always part of its window’s contents.

- A Windows scroll bar is usually a structural part of the window itself.

- When the user clicks in a window's scroll bar, a Macintosh program must hit-test the event and respond accordingly.
- To provide continuous action when the user holds down or drags the mouse inside a scroll bar, the Macintosh Toolbox repeatedly calls an action procedure supplied by the program.

- A Windows scroll bar does its own hit-testing and sends a scroll message to its parent window.
- To provide continuous action when the user holds down or drags the mouse inside a scroll bar, Windows sends repeated scroll messages to the window procedure of the scroll bar's parent window.

Chapter 8

Text

It's a rare program that doesn't have to deal with text charactes in some form or another. Windows supports this need with a predefined window class called *edit controls,* which support basic text operations like input, display, selection, and cut-and-paste editing—much like the original (unstyled) TextEdit routines of the Macintosh Toolbox. In this chapter, we'll learn how edit controls work, along with some other text-related topics like character sets, insertion carets, and general utility functions for working with text characters and strings.

Character Sets

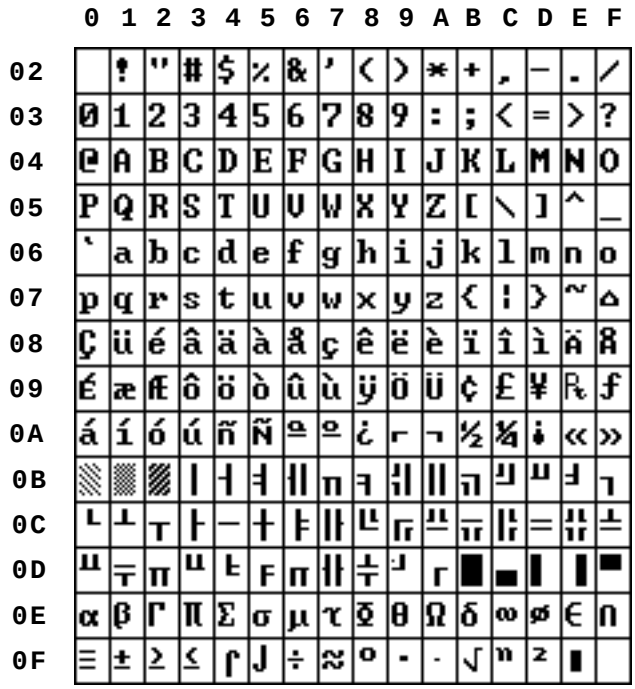
One of the conveniences that Macintosh programmers have always taken for granted is a stable, uniform character set. No matter what model of Macintosh your program is running on, from a 128K "Skinny Mac," vintage 1984, to a laptop Powerbook to a PowerMac 8100 with 16 MB of RAM and a 2-gigabyte hard disk, you can always confidently assume that the character code **`0x51`** stands for a capital **`Q`**, **`0xAD`** is a not-equal sign (≠), and **`0x8D`** is a lowercase **`c`** with a cedilla (**`ç`**). As in so many other cases, life is not so simple in the DOS/Windows world. Historically, programs running on PC-compatible machines have had to cope with the presence of two different character sets.

The OEM Character Set

When IBM's engineers were designing the original PC back in the early '80s, they knew they had to give it a character set based on the widely accepted industry standard, ASCII (American Standard Code for Information Interchange). But ASCII uses only 7 bits per character, 128 possible character codes in all; and even of those 128, 33 are assigned to nonprinting control functions and only 95 denote actual, printable characters. On a machine with an 8-bit byte, allowing 256 possible values, that left room for another 161 characters. So they scratched their heads a bit and came up with the character set shown in Figure 8–1.

Those three big blue initials guaranteed that this character set would be adopted by all of the rival PC-compatibles seeking to conform to the "IBM standard." A few variants did appear, mainly for use in foreign countries whose languages demanded additional accents and other special characters not included in the official IBM set. But in one form or another, virtually every PC-compatible computer sold included a native character set substantially similar to this one. Since this is the character set built into the computer by the manufacturer, it is known in Windows lingo as the *OEM character set* (for "original equipment manufacturer").

Figure 8–1. The IBM extended character set

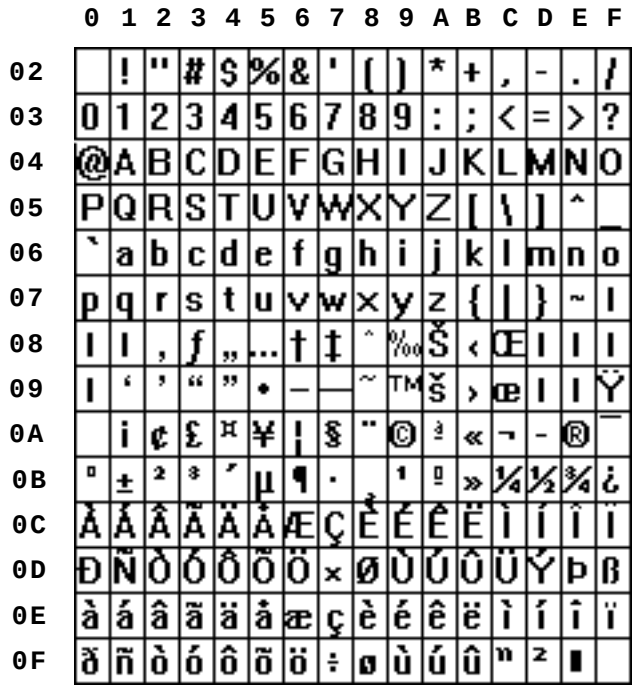


The ANSI Character Set

Perhaps the most striking feature of the IBM extended character set is its collection of line- and block-oriented graphical elements, included to allow graphical effects to be drawn on a fixed-pitch character screen of 80 columns by 24 rows. As time went on, the advent of true high-resolution bit-mapped screens made all those funny graphical characters unnecessary. So instead of using the OEM character set, early versions of Windows opted for the ANSI (American National Standards Institute) character set shown in Figure 8–2. Like the IBM version, ANSI is an 8-bit extended character set based on 7-bit ASCII, but with a more useful selection of accented, foreign, and special characters in place of the unneeded graphical elements. Windows has historically used this character set for all of its internal text handling, but still couldn't ignore the OEM set because—guess what?—the DOS file system used OEM for its directory and file names. So Windows programs had to be able to speak both dialects, converting between them if necessary with the Windows utility functions **`CharToOem`** and **`OemToChar`**.

With Windows 95, Windows has finally broken free of its dependence on the DOS file system. In general, programs running under Windows 95 can safely afford to ignore the OEM character set entirely. It's still supported for backward compatibility, though, and if your program is to run under older (pre–Windows 95) versions of the system, you may still need to be prepared to deal with OEM characters in places like file and path names.

Figure 8–2. The ANSI extended character set


0 1 2 3 4 5 6 7 8 9 A B C D E F
02 ! " # $ % & ' ( ) * + , - . /
03 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
04 @ A B C D E F G H I J K L M N O
05 P Q R S T U V W X Y Z [\ ] ^ _
06 ` a b c d e f g h i j k l m n o
07 p q r s t u v w x y z { | } ~
08 ‚ ƒ „ … † ‡ ˆ ‰ Š ‹ Œ
09 ‘ ’ “ ” • – — ˜ ™ š › œ Ÿ
0A ¡ ¢ £ ¤ ¥ ¦ § ¨ © ª « ¬ - ® ¯
0B ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿
0C À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï
0D Ð Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß
0E à á â ã ä å æ ç è é ê ë ì í î ï
0F ð ñ ò ó ô õ ö ÷ ø ù ú û ü ý

Character and String Operations

The Win32 interface includes a set of enhanced versions of standard C library functions for manipulating character strings. The enhanced versions all have names beginning with **`l`** (the lowercase letter “ell,” not the numeral “one”). For example, the Windows function **`lstrlen`** returns the length of a string in characters, like the standard C library function **`strlen`**. The Windows functions are designed to recognize and properly handle accented letters: for example, unlike the standard C library functions, the Windows versions will recognize the characters **`Ü`** and **`ü`** as upper- and lowercase versions of the same letter (u-umlaut). For the greatest flexibility, you should use enhanced Windows calls like **`lstrlen`** in your programs instead of standard C library calls like **`strlen`**. Table 8–1 lists the most common Windows utility functions for manipulating characters and character strings.

Table 8–1. Character manipulation functions

Function	Purpose
IsCharAlpha	Is character a letter?
IsCharAlphaNumeric	Is character a letter or digit?
IsCharUpper	Is character uppercase?
IsCharLower	Is character lowercase?
CharUpper	Convert to uppercase
CharLower	Convert to lowercase
CharToOem	Convert from ANSI to OEM
OemToChar	Convert from OEM to ANSI
lstrlen	String length
lstrcmp	Compare strings (case-sensitive)
lstrcmpi	Compare strings (case-insensitive)
lstrcpy	Copy string
lstrcat	Concatenate strings

Edit Controls

The basic Windows mechanism for working with text is the *edit control*. The real purpose for which edit controls were created was to serve as an editable text box in a dialog window, where the user can type in some sort of text for the dialog to operate on, such as a file name or a search string. In this sense, an edit record corresponds to a Macintosh dialog item of type **EditText**. Remember, though, that every Macintosh **EditText** item actually has a TextEdit record (**TERec**) associated with it for processing the user's input. The Windows edit control is actually analogous to this underlying TextEdit record, rather than just to the editable text item itself.

Like any control, an edit control actually a window—in this case, belonging to the predefined window class **EDIT**. Because it's only a control, it can't stand alone, but must be a child of some other window. Often the parent is a dialog window, but this is not a binding restriction: you can use an edit control in any kind of window. Our WiniEdit program, for instance, handles all of its text-editing operations through an edit control whose parent is an ordinary overlapped window.

On the other hand, because an edit control is a window in its own right, it can process its own mouse clicks, keystrokes, and other messages directly from the user or the system, without any intervention by the parent window. There is thus no need for Windows equivalents to Macintosh Toolbox routines like **TEKey**, for relaying keystrokes to a TextEdit record, **TEClick**, for relaying mouse clicks, and **TEUpdate**, for redrawing text on the screen in response to an update event. The built-in window procedure for class **EDIT** handles all of these operations on its own initiative by processing messages such as **WM_CHAR**, **WM_LBUTTONDOWN**, and **WM_PAINT**. An edit control can also have its own scroll bars, in which case it handles all of its own scrolling as well, again without any explicit action by the parent window. All of this makes working with edit controls quite a bit simpler than working with Macintosh TextEdit records.

Creating an Edit Control

You create an edit control in the usual way, by calling the Windows function **CreateWindow** (or **CreateWindowEx**), giving the string **"EDIT"** as the name of the window class. As always, one of the parameters you supply is a flag word containing style options to specify the new window's appearance and properties. Table 8–2 lists the available style options pertaining specifically to edit controls, all of which carry the prefix **ES_**, for "edit style." You can also include many of the standard window (**WS_**) styles that we learned about in Chapter 4: for instance, although an edit control, by default, has no visible border on the screen, you can give it one with the window style **WS_BORDER**. In particular, since an edit control is always a child of some other window, it should always have style **WS_CHILD**.

The standard form of edit control, intended specifically for use in dialog boxes, is a *single-line* edit control. As the name implies, it accepts just a single line of text from the user, suitable for typing in a file name or other item of information to the dialog. If the user presses the Enter key while typing text, it will be interpreted as a click of the dialog's default button (typically **OK**), rather than as starting a new line within the edit control. When the user's typing reaches the end of the edit box, the default behavior is to stop accepting additional characters and to send an error notification (**EN_MAXTEXT**) back to the parent window. You can override this, however, with the style option **ES_AUTOHSCROLL**, which causes the edit control to keep on accepting input from the keyboard and automatically scroll to the left to make room for it.

Style **ES_MULTILINE** creates a *multiline* edit control that can accommodate more than one line of text; this is the type of edit control that WiniEdit uses. If the control is part of a dialog box, however, the Enter key will still be taken to invoke the default button instead of starting a new line. The style option **ES_WANTRETURN** changes this behavior so that the Enter key simply begins a new line of text within the control. When the bottom edge is reached, the control will refuse any additional input and notify its parent with an **EN_MAXTEXT** notification—unless it

Table 8–2. Edit control style options

Style name	Meaning
ES_MULTILINE	Multiple lines of text
ES_WANTRETURN	Enter key starts new line instead of invoking default button
ES_AUTOVSCROLL	Automatic vertical scroll
ES_AUTOHSCROLL	Automatic horizontal scroll
ES_LEFT	Left-aligned text
ES_CENTER	Centered text
ES_RIGHT	Right-aligned text
ES_UPPERCASE	Force text to uppercase
ES_LOWERCASE	Force text to lowercase
ES_OEMCONVERT	Convert text from Windows to OEM character set
ES_READONLY	Read-only text: no input or editing allowed
ES_NOHIDESEL	Don't hide selection on losing input focus
ES_PASSWORD	Mask input with asterisks (*****) or other specified character

carries the style **ES_AUTOVSCROLL**, in which case it will simply scroll up one line and continue to accept text from the keyboard. In addition to the automatic scrolling options, you can give the edit control explicit, visible scroll bars of its own with the standard window styles **WS_VSCROLL** and **WS_HSCROLL**.

Other style options allow you to specify the alignment of text within the edit control (**ES_LEFT**, **ES_CENTER**, **ES_RIGHT**), convert it automatically to all upper- or all lowercase (**ES_UPPERCASE**, **ES_LOWERCASE**), or prevent the user from typing text into the control or editing its contents (**ES_READONLY**). By default, the control will normally highlight and unhighlight its selection automatically when it gains or loses the input focus; the **ES_NOHIDESEL** option causes the selection to stay highlighted even when the control doesn't have the focus. For working with file or directory pathnames under the old DOS file system (in older versions of Windows), style **ES_OEMCONVERT** automatically converts all characters from the ANSI to the OEM character set. Finally, for edit boxes that accept a security password of some kind from the user, there's the special-purpose style **ES_PASSWORD**. Instead of echoing the characters of the password back to the screen, where they can be seen by unauthorized eyes, this type of edit control masks them with a substitute character. The mask character is an asterisk (*****) by default, but you can change it by sending the edit control the message **EM_SETPASSWORDCHARACTER** or find out the current mask character with **EM_GETPASSWORDCHARACTER**.

Listing 8–1 shows how WiniEdit creates its edit control. This is a slightly simplified version of the program's **DoCreate** routine, called from the window procedure when it receives the message **WM_CREATE**. Recall that this is the first message Windows sends to a window after creating it. For WiniEdit's main document window, the main order of business at creation time is to create its child, the edit control. The style options specify that the edit control is a child, that it is initially visible when created, accommodates multiple lines of left-aligned text, has a vertical scroll bar, and scrolls automatically when typing reaches the bottom edge of the control. Other parameters to **CreateWindow** give the class name (**"EDIT"**), the parent window, and the owning program instance. The control's child identifier, for use in sending notification messages to its parent, is **Edit_Control**, defined as a constant in the program's header file, **WiniEdit.h**. Its title and coordinates are left unspecified, since it has no visible title bar and its location within the parent window will be set later. After creating the edit control, WiniEdit saves its handle for future use in a global variable named **TheEditor**.

After creating an edit control, you can just leave it empty waiting for the user to type text into it, or you can explicitly give it some initial text to display. The control holds its text in a buffer in memory. The general window message **WM_SETTEXT**, which ordinarily sets a window's title, copies text into this content buffer when sent to an edit control; similarly, the message **WM_GETTEXT** returns a copy of the buffer's current contents and **WM_GETTEXTLENGTH** returns the length of the contents in characters. The message **EM_SETHANDLE**, specific to edit controls, operates directly on the buffer handle, replacing it outright with a new buffer instead of copying text into the existing one; **EM_GETHANDLE** retrieves the current buffer handle.

Listing 8–1. Handle **WM_CREATE** message

```
VOID DoCreate (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_CREATE message.

   {
      DWORD     editStyle;                          // Style options for edit control


      editStyle = WS_CHILD |                        // Child window
                  WS_VISIBLE |                      //    visible on screen
                  WS_VSCROLL |                      //    vertical scroll bar
                  ES_AUTOVSCROLL |                  //    vertical autoscroll
                  ES_MULTILINE |                    //    multiple lines of text
                  ES_LEFT;                          //    flush-left alignment

      TheEditor = CreateWindow ("EDIT",             // Standard edit control
                                NULL,               // No title
                                editStyle,          // Style options as above
                                0, 0, 0, 0,         // Position and size will be set later
                                thisWindow,         // Main window is the parent
                                HMENU(Edit_Control),// Child identifier for notification messages
                                ThisInstance,       // Current program instance is the owner
                                NULL);              // No special creation parameters

   }  /* end DoCreate */
```

Text Layout

Like a Macintosh TextEdit record with its destination and view rectangles, a multiline edit control wraps text against a *formatting rectangle* and clips it to a *bounding rectangle.* Since the edit control is always a child of some other window, both rectangles are expressed in client coordinates, relative to the parent window's client area. The bounding rectangle is simply the boundary of the edit control's own client area, specified at creation time or set with the standard window functions **MoveWindow** or **SetWindowPos**. All text the edit control displays on the screen is clipped to the edges of this rectangle. The formatting rectangle, analogous to the destination rectangle on the Macintosh, defines the boundary against which text is wrapped into lines. The formatting rectangle is initially the same as the bounding rectangle, but you can change it with the message **EM_SETRECT**. This redefines the formatting rectangle, automatically rewraps the control's text to the new boundary, and redisplays the text on the screen. You can suppress the automatic redisplay by using the message **EM_SETRECTNP** instead; **EM_GETRECT** retrieves the current formatting rectangle.

WiniEdit's **DoSize** function (Listing 8–2) uses these facilities to adjust the size of its edit control to match that of the main document window. This function is called from WiniEdit's window procedure when it receives the message **WM_SIZE**, telling it that the main document window has been resized. The function extracts the new dimensions of the document window from the message parameters and calls the Windows function **MoveWindow** to set the edit

Listing 8–2. Handle **WM_SIZE** message

```
VOID DoSize (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_SIZE message.

   {
      INT     newWidth;                              // New width of client area
      INT     newHeight;                             // New height of client area
      RECT    textRect;                              // Formatting rectangle for wrapping text
      LPARAM  rectParam;                             // Pointer to rectangle as long-word parameter


      newWidth  = LOWORD(lParam);                    // Extract new dimensions
      newHeight = HIWORD(lParam);                    //    from message parameter

      MoveWindow (TheEditor,                         // Resize edit control to fit
                  0, 0,
                  newWidth, newHeight,
                  TRUE);

      rectParam = LPARAM(&textRect);                 // Convert to long integer
      SendMessage (TheEditor, EM_GETRECT, 0, rectParam); // Get formatting rectangle
      InflateRect (&textRect, -TextMargin, -TextMargin); // Inset by text margin
      SendMessage (TheEditor, EM_SETRECT, 0, rectParam); // Set new rectangle

   }  /* end DoSize */
```

control to these same dimensions. This explicitly sets the control's bounding rectangle and implicitly adjusts the formatting rectangle to match. Our **DoSize** function then retrieves the new formatting rectangle with the message **EM_GETRECT**, insets it by a small margin to provide some white space around the edges (using the Windows utility function **InflateRect**), and adjusts it to the smaller dimensions with the **EM_SETRECT** message. Text will now be wrapped to the smaller, inset formatting rectangle and displayed within the larger bounding rectangle, which coincides with the parent window's client area.

Once the edit control has wrapped its text to the formatting rectangle, you can find out the number of text lines by sending the message EM_GETLINECOUNT. Individual lines or characters are indexed from 0 for the first to $n - 1$ for the last, where n is the total number of lines or characters the control contains. The message **EM_LINEINDEX** returns the index of the first character in a given line, **EM_LINELENGTH** gives the number of characters in the line, and **EM_GETLINE** copies the entire contents of the line into a designated string buffer. In the other direction, **EM_LINEFROMCHAR** finds the number of the line containing a given character.

By default, Windows uses spaces and end-of-line characters (carriage returns and line feeds) to delimit words when wrapping text against the formatting rectangle. As on the Macintosh, you can modify this definition by providing your own word-break function to determine where a line may be broken. The messages for this purpose are **EM_SETWORDBREAKPROC** and **EM_GETWORDBREAKPROC**; the *Win32 Programmer's Reference* gives details on how to define the word-break procedure itself. Ordinarily, line breaks that are introduced as part of the word-wrapping process but not typed explicitly by the user are "soft," meaning that they appear on the screen but are not reflected in the text itself; with the message **EM_FMTLINES**, you can specify

that such line breaks be embedded directly in the text as sequences of explicit carriage-return and line-feed characters.

Text Selection

An edit control's *selection range* is expressed as a pair of integers representing the character positions at either end of the range. As on the Macintosh, you can think of these numbers as referring to the positions *between* the characters, rather than the characters themselves. The beginning of the text is position **0** and the end is equal to the length of the text in characters. A selection range from **10** to **20** includes the tenth through nineteenth characters in the text, but not the twentieth (since position **20** falls between the nineteenth and twentieth characters). If the beginning and ending positions are equal, the selection collapses to an empty insertion point, marked by a *caret* between characters of text.

The built-in window procedure for class **EDIT** processes the user's mouse clicks in an edit control's client area and uses them to drag out a new selection range. In the process, it automatically takes care of all of the needed details, such as highlighting the selection, extending or shortening it in response to a Shift-click, and selecting at the word instead of the character level on a double click. If the edit control has the **ES_AUTOVSCROLL** or **ES_AUTOHSCROLL** style (or both), it will scroll automatically in the indicated direction when the user drags the mouse outside the formatting rectangle while selecting.

The message **EM_GETSEL** returns an edit control's current selection range, while **EM_SETSEL** sets it. Specifying a range from **0** to **-1** selects the entire contents of the edit control. WiniEdit uses this feature to implement the **Select All** command on its **Edit** menu, as shown in Listing 8–3 (**DoSelectAll**).

As we mentioned earlier, an edit control will ordinarily highlight its selection or display its insertion caret on gaining the input focus and hide them on losing it. (The style option **ES_NOHIDESEL** forces the selection or caret to remain visible even when the control doesn't have the focus.) The control marks these transitions by sending the notification messages **EN_SETFOCUS** and **EN_KILLFOCUS** to its parent window.

Scrolling

In addition to scrolling automatically in the course of text input or selection, an edit control with scroll bars (specified by the style options **WS_VSCROLL** and **WS_HSCROLL**) handles all of the user's mouse actions affecting them and responds by scrolling its contents on the screen. The edit control alerts its parent window with an **EN_VSCROLL** or **EN_HSCROLL** notification before responding to the scroll request, allowing the parent to intervene if appropriate. The parent can

Listing 8–3. Handle **Select All** command

```
VOID DoSelectAll (VOID)

   // Handle Select All command.
   {
      SendMessage (TheEditor, EM_SETSEL, 0, -1);         // Select entire document

   }  /* end DoSelectAll */
```

also explicitly scroll the edit control's contents by sending it the message **EM_SCROLL** to scroll vertically by a single line or page, **EM_VSCROLL** to scroll by multiple units horizontally or vertically, or **EM_SCROLLCARET** to scroll the selection or insertion caret into view. The **EM_GETFIRSTVISIBLELINE** message returns the index of the first line of text currently visible in the edit control.

Editing Text

Edit controls support the standard cut-and-paste editing operations via the messages **WM_CUT**, **WM_COPY**, **WM_PASTE**, and **WM_CLEAR**. (They're classified as window messages, with the prefix **WM_** instead of **EM_**, because they can be sent to another form of control, combo boxes, as well as to edit controls.) **WM_CUT** removes the currently selected text from the edit control to the global clipboard, **WM_COPY** copies it to the clipboard without removing it from the edit control, **WM_CLEAR** removes it from the edit control without copying it to the clipboard, and **WM_PASTE** replaces it with the current contents of the clipboard. Another message, **EM_REPLACESEL**, replaces the current selection with a specified string of characters, rather than with the contents of the clipboard. WiniEdit uses these messages to implement the corresponding editing commands on its **Edit** menu by simply relaying them to the edit control for action. Listing 8–4 (**DoCut**) shows an example; the handling of the other editing commands is equally straightforward.

The **EDIT** window procedure also implements a built-in undo facility. The undo is only one level deep, meaning that only the immediately preceding editing operation can be undone: a second consecutive undo "undoes the undo," reinstating the results of the original operation. The message **EM_CANUNDO** reports whether the last editing operation can be undone; if so, **EM_UNDO** undoes it. Another related message, **EM_EMPTYUNDOBUFFER**, clears the edit control's internal undo buffer, rendering the undo operation unavailable.

Listing 8–4. Handle **Cut** command

```
VOID DoCut (VOID)

   // Handle Cut command.
   {
      SendMessage (TheEditor, WM_CUT, 0, 0);       // Relay operation to edit control

   }  /* end DoCut */
```

Table 8–3. Edit control messages

Message type	Mac counterpart	Meaning
`WM_GETTEXTLENGTH`	`TERec.teLength`	Get length of text in characters
`WM_GETTEXT`	`TEGetText`	Get text
`WM_SETTEXT`	`TESetText`	Set text
`EM_GETHANDLE`	`TERec.hText`	Get text handle
`EM_SETHANDLE`	`TERec.hText`	Set text handle
`EM_GETLINECOUNT`	`TERec.nLines`	Get number of text lines
`EM_LINELENGTH`	————	Get number of characters in given line
`EM_GETLINE`	————	Copy contents of specified text line
`EM_LINEINDEX`	`TERec.lineStarts`	Get index of first character in given line
`EM_LINEFROMCHAR`	————	Get index of line containing given character
`EM_SCROLL`	`TEScroll`	Scroll text vertically
`EM_LINESCROLL`	`TEScroll`	Scroll text vertically or horizontally or both
`EM_SCROLLCARET`	`TESelView`	Scroll insertion caret into view
`EM_GETFIRSTVISIBLELINE`	————	Get index of first visible line
`EM_GETSEL`	`TERec.selStart`, `TERec.selEnd`	Get selection range
`EM_SETSEL`	`TESetSelect`	Set selection range
`EM_REPLACESEL`	`TEInsert`	Replace current selection with specified text
`WM_CUT`	`TECut`	Cut current selection to paste buffer
`WM_COPY`	`TECopy`	Copy current selection to paste buffer
`WM_PASTE`	`TEPaste`	Replace current selection with paste buffer
`WM_CLEAR`	`TEDelete`	Clear current selection
`EM_CANUNDO`	————	Can last operation be undone?
`EM_UNDO`	————	Undo last operation
`EM_EMPTYUNDOBUFFER`	————	Disable undo of last operation
`EM_GETMODIFY`	————	Has text been modified?
`EM_SETMODIFY`	————	Mark or unmark text as modified
`EM_SETREADONLY`	————	Set or clear read-only style
`EM_SETTABSTOPS`	————	Set tab positions
`EM_GETRECT`	`TERec.destRect`	Get formatting rectangle
`EM_SETRECT`	`TERec.destRect`	Set formatting rectangle
`EM_SETRECTNP`	`TERec.destRect`	Set formatting rectangle without redrawing
`EM_FMTLINES`	————	Insert hard line breaks when word-wrapping?
`EM_GETWORDBREAKPROC`	`TERec.wordBreak`	Get word-break function
`EM_SETWORDBREAKPROC`	`SetWordBreak`	Set word-break function
`EM_GETPASSWORDCHAR`	————	Get mask character for hiding passwords
`EM_SETPASSWORDCHAR`	————	Set mask character for hiding passwords

Table 8–4. Edit control notification messages

Notification code	Meaning
EN_CHANGE	Text contents changed
EN_UPDATE	About to redisplay text
EN_VSCROLL	About to scroll vertically
EN_HSCROLL	About to scroll horizontally
EN_SETFOCUS	Gaining input focus
EN_KILLFOCUS	Relinquishing input focus
EN_MAXTEXT	Text capacity exceeded
EN_ERRSPACE	Out of space

An edit control maintains a flag telling whether its contents are currently "dirty"—that is, have been modified and not yet saved. This *modify flag* is initially cleared to **FALSE** when the edit control is created, and automatically set to **TRUE** whenever any editing or input operation alters the contents of the control. You can also set the state of the flag explicitly, either **TRUE** or **FALSE**, with the message **EM_SETMODIFY**. WiniEdit uses this message, for example, to clear the flag (marking the text as clean) whenever it saves the contents of its window to a file, reads in a new file, or reverts to a previously saved version of a file. It then uses the companion message **EM_GETMODIFY** to check the state of the flag before displaying its **File** menu and decide whether to enable or gray out the **Save** and **Revert to Saved...** commands.

In addition to setting the modify flag, an edit control also sends the notification **EN_CHANGE** to its parent window whenever the contents of its text change in any way; if you need to, you can use this notification to track the changes on the fly instead of polling for them after the fact with **EM_GETMODIFY**. Another notification, **EN_UPDATE**, is sent just before redisplaying the text on the screen, giving you a chance to do any preprocessing or adjustment you may need at that point. Tables 8–3 and 8–4 summarize the control messages an edit control accepts and the notifications it sends.

The Caret

When a window has the input focus, it can display a *caret* to mark the insertion point for text or graphics. In particular, an edit control automatically displays an insertion caret whenever its selection range is empty (provided, of course, that the insertion point is not scrolled out of view in the window's client area). The caret is not limited to edit controls, however: any window can display one to show that it is prepared to accept input from the user. Table 8–5 lists the available Windows functions for working with the caret.

An important thing to remember about the caret is that it is a shared resource. There isn't a separate caret for each window: there is one and only one for the entire system. Only one window at a time, the window with the current input focus, can own the caret. A window should not attempt to manipulate the caret, then, unless it has the focus. Typically, a window sets the caret's appearance on gaining the input focus and relinquishes it on losing the focus, by calling the

Table 8–5. Caret functions

Function	Purpose
`CreateCaret`	Set caret shape
`DestroyCaret`	Remove caret shape
`ShowCaret`	Display caret
`HideCaret`	Hide caret
`GetCaretPos`	Get caret position
`SetCaretPos`	Set caret position
`GetCaretBlinkTime`	Get caret blink interval
`SetCaretBlinkTime`	Set caret blink interval

Windows functions **`CreateCaret`** and **`DestroyCaret`** while processing the messages **`WM_SETFOCUS`** and **`WM_KILLFOCUS`**, respectively.

`CreateCaret` can set the caret to either a solid or gray rectangle of specified dimensions, or to some other graphical appearance defined by a bitmap supplied as a parameter. The typical caret for text insertion is a solid rectangle as high as a text character and the same width as the standard window border. **`CreateCaret`** will use this border width automatically if you default the width parameter to **`0`**, allowing the caret to scale with the resolution of the screen so that it will appear reasonable on any display device. There's normally no need for you to specify this value explicitly, but you can find it out if you need to by calling the **`GetSystemMetrics`** function with a selector of **`SM_CXBORDER`**. The caret blinks on the screen at a rate you can learn with the Windows function **`GetCaretBlinkTime`** or set with **`SetCaretBlinkTime`**.

The **`GetCaretPos`** and **`SetCaretPos`** functions retrieve and set the caret's position within a window, expressed in coordinates relative to the window's client area. **`HideCaret`** makes the caret invisible and **`ShowCaret`** displays it again. These two functions work by decrementing and incrementing a visibility count similar to the one that controls the cursor, so they must be balanced for the caret to be visible: that is, if you hide the caret three times in a row, you must show it three times before it will reappear on the screen. To keep the caret from being inadvertently trashed when a window's contents are repainted, the **`BeginPaint`** and **`EndPaint`** functions hide it before the repaint and show it again afterward. This is one reason for always using these functions when responding to a **`WM_PAINT`** message.

The Same Thing...

- The Macintosh uses TextEdit records for onscreen text entry and manipulation.
- A Macintosh TextEdit record wraps text to a destination rectangle and clips it to a view rectangle.
- Windows uses edit controls for onscreen text entry and manipulation.
- A Windows edit control wraps text to a formatting rectangle and clips it to a bounding rectangle.

• A Macintosh TextEdit record has a word-break function that can customize the way it wraps text to its destination rectangle.	• A Windows edit control has a word-break function that can customize the way it wraps text to its formatting rectangle.
• A Macintosh TextEdit record has a selection range expressed as a starting and ending character position.	• A Windows edit control has a selection range expressed as a starting and ending character position.
• A Macintosh TextEdit record supports the standard editing operations Cut, Copy, Paste, and Clear.	• A Windows edit control supports the standard editing operations Cut, Copy, Paste, and Clear.

...Only Different

• The Macintosh uses a single, standard 8-bit character set.	• Windows has historically used two different 8-bit character sets: OEM (original equipment manufacturer) and ANSI (American National Standards Institute).
• A Macintosh TextEdit record must have its keystrokes, mouse clicks, and update events relayed to it from the main event loop via the Toolbox routines **`TEKey`**, **`TEClick`**, and **`TEUpdate`**.	• A Windows edit control has its own window procedure and can process its own keystrokes, mouse clicks, and repaint messages.
• A Macintosh TextEdit record is not directly connected to a scroll bar; mouse clicks in the scroll bar must be explicitly relayed to the TextEdit record via the Toolbox routine **`TEScroll`**.	• A Windows edit control has its own scroll bars and handles its own scrolling.
• A Macintosh TextEdit record must be told explicitly to display and hide its selection with the Toolbox routines **`TEActivate`** and **`TEDeactivate`**.	• A Windows edit control displays and hides its selection automatically on gaining and losing the input focus.
• A Macintosh TextEdit record doesn't support the Undo operation.	• A Windows edit control supports the Undo operation.
• A Macintosh TextEdit record must be told to blink its insertion caret via the Toolbox routine **`TEIdle`**.	• A Windows edit control blinks its insertion caret automatically.

Chapter 9

Menus

Menus are as important to the Windows user interface as they are on the Macintosh. On both systems, they are the principal means by which the user can issue commands and direct the actions of the program. With a few minor differences, Windows and Macintosh menus look and feel pretty much the same to the user: they pop up when you click their title in the menu bar, offering a set of actions or options from which to choose by pointing with the mouse. From the programmer's point of view, menus are integrated into the Windows message mechanism, simplifying the task of decoding and responding to the user's menu choices. In this chapter, we'll learn the basics of how Windows menus work and how to handle them in your programs.

Menu Basics

Windows menus come in two main flavors. A *top-level menu* corresponds to the Macintosh menu bar: a horizontal series of text items that typically represent the titles of individual menus (see Figure 9–1). In fact, a top-level menu is often referred to informally in Windows as a "menu bar," though it's actually just a special type of menu that happens to be displayed horizontally on the screen instead of vertically. Note that, unlike its Macintosh counterpart, a top-level menu always belongs to a particular window and appears below that window's title bar, rather than globally at the top of the screen. Only a top-level window—not a child—can have a menu bar. Not every window has to have a menu bar, but every menu bar must belong to a window.

What Macintosh programmers normally think of as a menu is called a *pop-up menu* (or *submenu*) in Windows (Figure 9–2). This type of menu remains hidden from view until the user clicks the mouse in its title. The menu then "pops up" on the screen, allowing the user to choose one of its items with the mouse. Unlike those on the Macintosh, a Windows menu remains on the screen even after the user releases the mouse button: it isn't necessary to hold down the button continuously in order to keep the menu visible. If the user rolls the mouse along the menu bar to another menu's title, the first menu will disappear and the second menu will pop up in its place, even without a further mouse click. The user must click the mouse a second time to make the menu go away: either by clicking in one of its items (choosing that item) or by clicking anywhere outside the menu (dismissing it without choosing an item).

Figure 9–1. A top-level menu

Figure 9–2. A pop-up menu

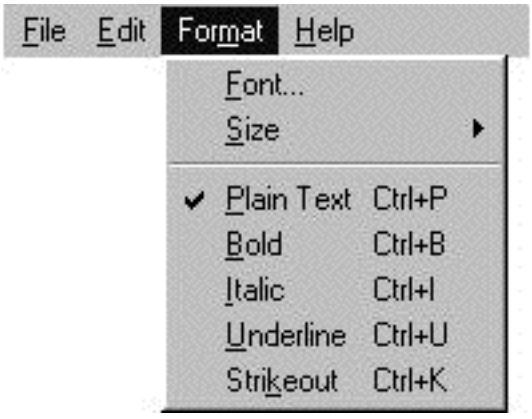


Figure 9–2 shows the three types of item can appear on a menu:

- A *command item* invokes an action of some sort by issuing a message to the window procedure of the window that owns the menu.
- A *pop-up item* is actually the title of another menu; choosing it with the mouse causes the associated menu to pop up on the screen.
- A *separator* is a horizontal line used to separate the menu's items into groups for easier readability. It is not an active item and does not respond to the mouse.

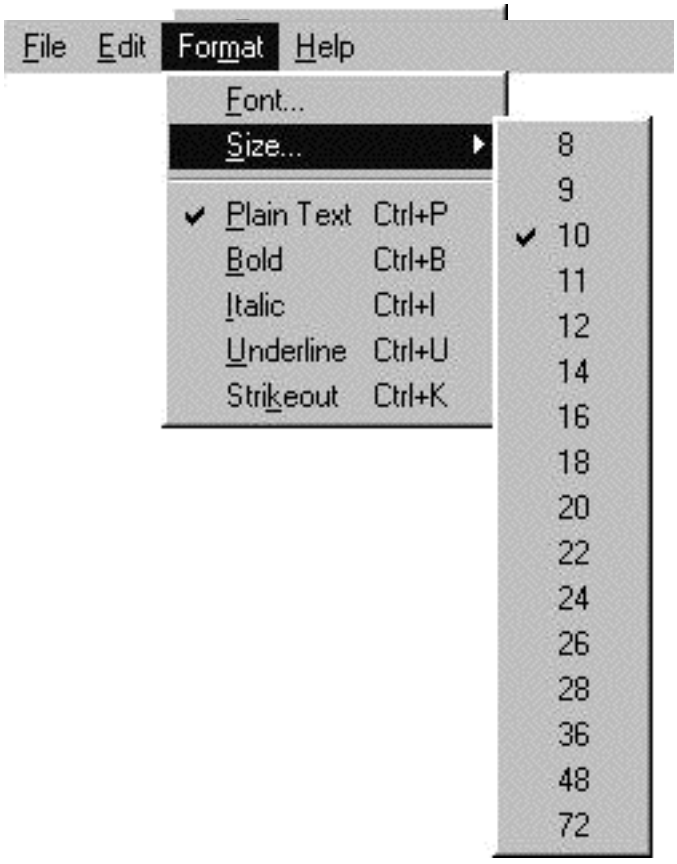
Only a pop-up menu can contain separators, but the other two types can appear on both top-level and pop-up menus. In the menu bar, pop-up items work just like the menu titles in the Macintosh menu bar: they display a pop-up menu that hangs down below the title, as shown in Figure 9–2. In pop-up menus, they work like hierarchical menus on the Macintosh, bringing up a submenu to the right of the original menu (Figure 9–3); such hierarchical submenus can be nested to any depth. As on the Macintosh, a right-pointing triangle symbol to the right of the item cues the user that the item invokes a submenu rather than a command. One difference from the Macintosh is that command items, too, can appear in both top-level and pop-up menus. Thus, although the menu bar ordinarily contains nothing but pop-up menu titles, it can also, in unusual cases, contain items that activate an immediate command instead of a menu.

Individual menu items can be placed in any of three states:

- An *enabled* item can be chosen with the mouse and will respond by invoking a command or displaying a submenu, as the case may be.
- A *disabled* item is displayed in the normal way, but does not respond to the mouse.
- A *grayed* item is displayed in gray instead of black and does not respond to the mouse.

By default, all menu items are normally enabled. As on the Macintosh, you should gray out any items that are temporarily inappropriate or inapplicable in a given situtation, such as a **`Cut`** or **`Copy`** command when there is no current selection or a **`Save`** command when the window doesn't contain any unsaved changes. Disabling is more appropriate for items that are intended to be permanently inactive, such as a title introducing a group of related items on the menu. Grayed items highlight in the normal way when the user points to them with the mouse, but cause no

Figure 9–4. The system menu
Figure 9–3. A hierarchical pop-up menu



action to occur if the mouse is clicked; disabled items are completely inert and do not even highlight when the mouse rolls over them. The Windows function **`EnableMenuItem`** sets the state of a menu item enables, disables, or grays a menu item.

As on the Macintosh, a menu item can be marked with a check mark, or *checked*, to show that it is currently in effect (see Figure 9–2). You can use the Windows function **`CheckMenuItem`** to check or uncheck an item and **`GetMenuState`** to find out whether it is currently checked or unchecked. The standard check mark symbol shown in the figure is used by default, but it is possible to substitute a custom symbol of your own with a Windows function called **`SetMenuItemBitmaps`**.

One particularly important pop-up menu is the *system menu* (Figure 9–4), which appears when the user clicks the small program icon at the left of a window's title bar. The system menu typically lists commands for manipulating the window on the screen, such as **`Move`**, **`Size`**, **`Minimize`**, **`Maximize`**, **`Restore`**, and **`Close`**, which are normally handled by the Windows system itself through its default window procedure. A program is free to intercept and respond to these commands itself, or to add further commands of its own to the system menu, but most programs just let the system handle all interactions with this menu in its own way.

Not every pop-up menu has to be anchored to a pop-up item in the menu bar or another menu. It's also possible to create a *floating pop-up menu* that pops up right at the mouse location in response to some action by the user, typically a click of the right mouse button. The contents of the

resulting menu can vary depending on the context, circumstances, and area of the screen in which the mouse is pressed: for example, many programs display the system menu as a floating pop-up when the user clicks the right button in a window's title bar.

Creating and Destroying Menus

As on the Macintosh, you can either build your menus "by hand," using system functions provided for the purpose, or load them from the disk as resources. To build a menu from scratch, you begin by calling either the Windows function **`CreateMenu`** to create an empty menu bar, or **`CreatePopupMenu`** to create an empty pop-up menu. Both functions return a result of type **`HMENU`**, a handle to a menu. You can then add items to the menu one by one, using either **`AppendMenu`** to add a new item at the end of the menu or **`InsertMenu`** to insert it before another specified item.

One of the parameters you supply to both **`AppendMenu`** and **`InsertMenu`** is a *menu item identifier*, a 16-bit integer that will uniquely identify the item in all communications between your program and the Windows system. Item identifiers are purely arbitrary numbers that you define for yourself: they aren't assigned automatically by the system, nor do they bear any necessary relation to the item's position on the menu. Listing 9–1 shows the item identifiers used by our WiniEdit program, taken from the header file **`WiniEdit Resources.h`**.

Listing 9–1. WiniEdit menu-item identifiers

```
#define Main_Menu                       1000

#define File_Menu                   0
#define    New_Item                     1001
#define    Open_Item                    1002
#define    Close_Item                   1003
#define    Save_Item                    1004
#define    SaveAs_Item                  1005
#define    Revert_Item                  1006
#define    Setup_Item                   1007
#define    Print_Item                   1008
#define    Exit_Item                    1009

#define Edit_Menu                   1
#define    Undo_Item                    1101
#define    Cut_Item                     1102
#define    Copy_Item                    1103
#define    Paste_Item                   1104
#define    Delete_Item                  1105
#define    SelectAll_Item               1106

#define Format_Menu                 2
#define    Format_Item                  1201
#define    Default_Item                 1202
#define    Background_Item              1203

#define Help_Menu                   3
#define    Help_Item                    1301
#define    About_Item                   1302
```

Besides the item identifier, both **`AppendMenu`** and **`InsertMenu`** accept a menu handle identifying the menu to which the new item is to be added, along with a flag word describing the item's characteristics, as shown in Table 9–1. **`InsertMenu`** takes an additional parameter specifying where in the menu the new item is to be inserted. Finally, both functions accept a parameter defining the content of the new menu item, depending on its type: a character string for a text item, a bitmap for a graphical item, or a 32-bit data value to be passed to the drawing routine for a custom item.

Although you can certainly build all your menus by hand if you want to, it's generally a better idea to read them in as resources. A single *menu template* resource defines a top-level menu and all of its submenus, filling the roles of both **`'MBAR'`** and **`'MENU'`** resources on the Macintosh. I built all of WiniEdit's menus directly onscreen, using the interactive menu editor from the Visual C++ development environment. The editor produced the resource description shown in Listing 9–2, in the description language expected by the Visual C++ resource compiler. The resource compiler then compiled the description into a menu template that can be read into memory with the Windows function **`LoadMenu`**.

Table 9–1. Menu-item style options

Style name	Meaning
MF_STRING	Text item
MF_BITMAP	Graphical item
MF_OWNERDRAW	Custom item (drawn by application)
MF_POPUP	Pop-up item (invokes another menu)
MF_SEPARATOR	Separator line
MF_ENABLED	Item is enabled
MF_DISABLED	Item is disabled
MF_GRAYED	Item is grayed
MF_CHECKED	Item is checked
MF_UNCHECKED	Item is unchecked
MF_MENUBREAK	Item starts new line (in menu bar) or new column (in pop-up menu)
MF_MENUBARBREAK	Same as **MF_MENUBREAK**, but with vertical separator line between columns in pop-up menu

Once you've created a top-level menu, you have to associate it with a window. There are three ways of doing this:

- Make it the default menu for the window's class. You do this by storing the menu's resource name or ID into the **lpszMenuName** field of the **WNDCLASS** structure that you pass to **RegisterClass**. This is how WiniEdit does it, in its **InitProgram** routine:

```
resourceID = MAKEINTRESOURCE(Main_Menu);     // Convert resource ID to string
windowClass.lpszMenuName = resourceID;       // Set menu
```

- Specify it as a parameter at the time the window is created. To do this, you must first create the menu and obtain a handle to it, either by building it from scratch with **CreateMenu**, **AppendMenu**, and **InsertMenu** or by loading it from a template resource with **LoadMenu**. Once you have a handle to an existing menu, you can pass it as a parameter to **CreateWindow**; if this parameter is non-null, it overrides the default menu associated with the window's class.
- Associate it directly with an existing window with the **SetMenu** function.

You can obtain a handle to a window's top-level menu with **GetMenu**, to a pop-up menu with **GetSubMenu**, or to the system menu with **GetSystemMenu**. Destroying a window automatically destroys its menu (including all submenus) along with it, so there's no need to dispose of the menu explicitly. A menu not associated with any window, such as a floating pop-up, must be destroyed with **DestroyMenu** when you're through with it; otherwise it will continue to take up space in memory even after your program terminates.

Listing 9–2. WiniEdit menu resource description

```
Main_Menu MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New\tCtrl+N",                New_Item
        MENUITEM "&Open...\tCtrl+O",            Open_Item
        MENUITEM "&Close\tCtrl+W",              Close_Item
        MENUITEM SEPARATOR
        MENUITEM "&Save\tCtrl+S",               Save_Item
        MENUITEM "Save &As...\tCtrl+Alt+S",     SaveAs_Item
        MENUITEM "&Revert to Saved...\tCtrl+R", Revert_Item
        MENUITEM SEPARATOR
        MENUITEM "Page Set&up...\tCtrl+Alt+P",  Setup_Item
        MENUITEM "&Print...\tCtrl+P",           Print_Item
        MENUITEM SEPARATOR
        MENUITEM "E&xit\tCtrl+Q",               Exit_Item
    END
    POPUP "&Edit"
    BEGIN
        MENUITEM "&Undo\tCtrl+Z",               Undo_Item
        MENUITEM SEPARATOR
        MENUITEM "Cu&t\tCtrl+X",                Cut_Item
        MENUITEM "&Copy\tCtrl+C",               Copy_Item
        MENUITEM "&Paste\tCtrl+V",              Paste_Item
        MENUITEM "&Delete\tDelete",             Delete_Item
        MENUITEM SEPARATOR
        MENUITEM "Select &All\tCtrl+A",         SelectAll_Item
    END
    POPUP "For&mat"
    BEGIN
        MENUITEM "Text &Format...\tCtrl+F",     Format_Item
        MENUITEM "&Default Format\tCtrl+D",     Default_Item
        MENUITEM SEPARATOR
        MENUITEM "&Background Color...\tCtrl+B", Background_Item
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&Help\tCtrl+?",               Help_Item
        MENUITEM SEPARATOR
        MENUITEM "&About WiniEdit...",          About_Item
    END
END
```

Menu Messages

Table 9–2 summarizes the menu-related messages that a window can receive. When the user clicks the mouse in the window's menu bar, Windows sends the message **WM_ENTERMENULOOP** to begin tracking the mouse for menu selection (roughly analogous to the Macintosh Toolbox's **MenuSelect** routine). This message allows you to get control at the beginning of the mouse-tracking loop in case you want to customize the menu's mouse-tracking behavior in some way. Theoretically, you could intercept this message and provide your own tracking loop; but in reality, you will almost certainly pass it to the default window procedure to track the mouse in the standard way. A companion message, **WM_EXITMENULOOP**, serves as a tail hook in case you want to do some kind of final customization after the button is released; again, most real-world programs simply ignore this message.

As the mouse rolls over each item in the menu, Windows sends a **WM_MENUSELECT** message for the item. The default window procedure responds by highlighting the item on the screen and, if it's a pop-up item, displaying the submenu associated with it. (Remember that the menu bar itself is the top-level menu, so even the first tier of pop-ups, the ones whose titles appear in the menu bar, are considered submenus.) You can intercept this message and customize the highlighting, if you like, or use the occasion to provide some other form of useful feedback such as a brief command description in a "status bar" at the bottom of your window.

Before displaying a pop-up menu, Windows sends you the message **WM_INITMENUPOPUP**. This gives you a chance to enable or gray out items on the menu according to the current context and circumstances, before they're presented to the user. The message's parameters identify the menu, the pop-up item that invoked it, and whether it's the system menu or one of your own. (A similar message, **WM_INITMENU**, allows you to modify the contents of the menu bar itself at the very beginning of the tracking loop, when the mouse button is first pressed in it—but it's hard to think of a convincing way to use it.)

Table 9–2. Menu-related messages

Message type	Meaning
WM_INITMENU	Top-level menu becoming active
WM_INITMENUPOPUP	Pop-up menu about to be displayed
WM_MEASUREITEM	Find dimensions of custom menu item
WM_DRAWITEM	Draw custom menu item
WM_ENTERMENULOOP	Entering menu tracking loop
WM_EXITMENULOOP	Leaving menu tracking loop
WM_MENUCHAR	Unknown keyboard shortcut received
WM_MENUSELECT	Menu item selected
WM_COMMAND	Menu command chosen
WM_SYSCOMMAND	System menu command chosen

Listing 9–3. Handle `WM_INITMENUPOPUP` message

```
VOID DoInitMenuPopup (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_INITMENUPOPUP message.

   {
      HMENU  theMenu   = HMENU(wParam);      // Handle to menu to be adjusted
      UINT   menuIndex = LOWORD(lParam);     // Relative position of menu in menu bar
      BOOL   isSystem  = HIWORD(lParam);     // Is it the system menu?


      if ( isSystem )                        // Is it the system menu?
         FixSystemMenu ();                   // Enable/disable system menu commands
      else
         switch ( menuIndex )
            {
               case File_Menu:
                  FixFileMenu (theMenu);     // Enable/disable File menu commands
                  break;

               case Edit_Menu:
                  FixEditMenu (theMenu);     // Enable/disable Edit menu commands
                  break;

               case Format_Menu:
                  FixFormatMenu (theMenu);   // Enable/disable Format menu commands
                  break;

               case Help_Menu:
                  FixHelpMenu (theMenu);     // Enable/disable Help menu commands
                  break;

            }  /* end switch ( menuIndex ) */

   }  /* end DoInitMenuPopup */
```

The parameters to the **`WM_INITMENUPOPUP`** message identify the menu being displayed, the item that invoked it, and whether it's the system menu or one of your own. WiniEdit's routine for handling this message, **`DoInitMenuPopup`** (Listing 9–3), simply extracts the identifying information from the parameters and then dispatches on the item ID to a specialized routine for initializing that particular menu. Listing 9–4 shows one of these routines, **`FixEditMenu`**, for illustration. This routine asks the window's edit control for the length of the text displayed in the window, the length of the current selection, and whether the last editing operation can be undone. It uses this information to decide whether to enable or gray the **`Undo`**, **`Cut`**, **`Copy`**, **`Delete`**, and **`Select All`** commands; for the **`Paste`** command, it checks for the availability of text data on the clipboard. Other WiniEdit routines perform the equivalent task of enabling and graying items, according to current conditions, on the **`File`**, **`Format`**, and **`Help`** menus.

Listing 9–4. Enable/disable **`Edit`** menu commands

```
VOID FixEditMenu (HMENU theMenu)

   // Enable/disable Edit menu commands.

   {
      BOOL  canUndo;                                          // Can editor support Undo command?
      LONG  selStart;                                         // Character position at start of selection
      LONG  selEnd;                                           // Character position at end of selection
      BOOL  canPaste;                                         // Is there text on the clipboard?
      LONG  textLength;                                       // Number of characters in document


      canUndo = SendMessage(TheEditor, EM_CANUNDO, 0, 0);     // Is Undo available?
      if ( canUndo )
         EnableMenuItem (theMenu, Undo_Item, MF_ENABLED);     // Enable Undo command
      else
         EnableMenuItem (theMenu, Undo_Item, MF_GRAYED);      // Gray out Undo command


      SendMessage ( TheEditor, EM_GETSEL,                     // Get selection range
                    WPARAM(&selStart), LPARAM(&selEnd) );
      if ( selStart != selEnd )                               // Is there a selection?
         {
            EnableMenuItem (theMenu, Cut_Item,    MF_ENABLED);    // Enable Cut command
            EnableMenuItem (theMenu, Copy_Item,   MF_ENABLED);    // Enable Copy command
            EnableMenuItem (theMenu, Delete_Item, MF_ENABLED);    // Enable Delete command

         }  /* end if ( selStart == selEnd ) */
      else
         {
            EnableMenuItem (theMenu, Cut_Item,    MF_GRAYED);     // Gray out Cut command
            EnableMenuItem (theMenu, Copy_Item,   MF_GRAYED);     // Gray out Copy command
            EnableMenuItem (theMenu, Delete_Item, MF_GRAYED);     // Gray out Delete command

         }  /* end else */


      canPaste = IsClipboardFormatAvailable(CF_TEXT);         // Does clipboard contain text?
      if ( canPaste )
         EnableMenuItem (theMenu, Paste_Item, MF_ENABLED);    // Enable Paste command
      else
         EnableMenuItem (theMenu, Paste_Item, MF_GRAYED);     // Gray out Paste command


      textLength = GetWindowTextLength(TheEditor);            // Get length of text
      if ( textLength > 0 )                                   // Any text in document?
         EnableMenuItem (theMenu, SelectAll_Item, MF_ENABLED);    // Enable Select All command
      else
         EnableMenuItem (theMenu, SelectAll_Item, MF_GRAYED);     // Gray out Select All command


   }  /* end FixEditMenu */
```

Two more menu messages, **WM_MEASUREITEM** and **WM_DRAWITEM**, are used for drawing custom menu items. (You may recall from Chapter 7 that these same two messages are also used for custom controls.) Most typical menus consist entirely of text items, which Windows already knows how to draw. Graphical items also present no problem, since they carry their own bitmaps with them. If you want to use custom items in your menus, however, then of course it's up to you to draw them yourself. Before drawing such an item for the first time, Windows sends you the message **WM_MEASUREITEM**, asking you to supply the item's dimensions so it knows how much space to reserve on the menu. Then, each time the item has to be drawn, Windows sends you a **WM_DRAWITEM** message. One of the parameters to this message is a data structure containing all the information needed to draw the item, including a handle to a device context. Your window procedure must respond to the message by drawing the item in the given context. Customizing menu items is an unusual thing for a program to do, and we won't spend any time on it here; if you're interested, you can find more information in the *Win32 Programmer's Reference.*

Of course, the most important menu message of all is **WM_COMMAND**, which tells you that the user has chosen a command item from one of your menus. Much of the real work of a program gets done in response to this message. (Another message, **WM_SYSCOMMAND**, reports that an item has been chosen from the system menu, rather than one of your own. You'll usually just want to let the default window procedure handle this one; normally, the only reason for processing it yourself would be if you've added one or more commands of your own to the system menu.)

Listing 9–5. Handle **WM_COMMAND** message

```
VOID DoCommand (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_COMMAND message.

   {
      UINT  notifyCode = HIWORD(wParam);                    // Notification code from child control
      UINT  itemID     = LOWORD(wParam);                    // Item ID of message originator
      HWND  theControl = HWND(lParam);                      // Handle to control sending notification


      switch ( notifyCode )
         {
            case 0:
            case 1:
               DoMenuCommand (itemID);                      // Handle menu command
               break;

            default:
               DoNotification (itemID, theControl, notifyCode);   // Handle control notification
               break;

         }  /* end switch ( notifyCode ) */

   }  /* end DoCommand */
```

Listing 9–6. Handle menu command

```
VOID DoMenuCommand (UINT itemID)

   // Handle menu command.

   {
      switch ( itemID )
         {

            /* File menu */

               case New_Item:
                  DoNew ();                    // Handle New command
                  break;

               case Open_Item:
                  DoOpen ();                   // Handle Open... command
                  break;

               case Close_Item:
                  DoClose ();                  // Handle Close command
                  break;

               case Save_Item:
                  DoSave ();                   // Handle Save command
                  break;

               case SaveAs_Item:
                  DoSaveAs ();                 // Handle Save As... command
                  break;

               case Revert_Item:
                  DoRevert ();                 // Handle Revert to Saved... command
                  break;

               case Setup_Item:
                  DoSetup ();                  // Handle Page Setup... command
                  break;

               case Print_Item:
                  DoPrint ();                  // Handle Print... command
                  break;

               case Exit_Item:
                  DoExit ();                   // Handle Exit command
                  break;

            /* Edit menu */

               case Undo_Item:
                  DoUndo ();                   // Handle Undo command
                  break;
```

Listing 9–6. Handle menu command (*continued*)

```
            case Cut_Item:
               DoCut ();                    // Handle Cut command
               break;

            case Copy_Item:
               DoCopy ();                   // Handle Copy command
               break;

            case Paste_Item:
               DoPaste ();                  // Handle Paste command
               break;

            case Delete_Item:
               DoDelete ();                 // Handle Delete command
               break;

            case SelectAll_Item:
               DoSelectAll ();              // Handle Select All command
               break;

         /* Format menu */

            case Format_Item:
               DoFormat ();                 // Handle Text Format... command
               break;

            case Default_Item:
               DoDefaultFormat ();          // Handle Default Format command
               break;

            case Background_Item:
               DoBackground ();             // Handle Background Color... command
               break;

         /* Help menu */

            case Help_Item:
               DoHelp ();                   // Handle Help command
               break;

            case About_Item:
               DoAbout ();                  // Handle About WiniEdit... command
               break;

            default:
               MessageBeep (MB_OK);         // Error: control should never
               break;                       //    reach this point

         }  /* end switch ( itemID ) */

}  /* end DoMenuCommand */
```

We've already encountered the **WM_COMMAND** message in Chapter 7, where it was used to report events affecting a control to the control's parent window. Recall that, when the message comes from a control, the high-order word of the **wParam** parameter contains a notification code identifying the nature of the event being reported. If the notification code is **0**, the message is from a menu item instead of a control. In this case, the low-order word of **wParam** gives the item identifier for the item that was chosen; **lParam** is irrelevant and is ignored. (A notification code of **1** denotes a menu item invoked from the keyboard rather than with the mouse; we'll be discussing this case in the next section.)

When WiniEdit's window procedure, **DoMessage**, receives a **WM_COMMAND** message, it calls the **DoCommand** routine shown in Listing 9–5. This routine simply unpacks the identifying information from the message parameters, examines the notification code to see if it's a menu command or a control notification, and calls the applicable WiniEdit routine, either **DoMenuCommand** or **DoNotification**. **DoMenuCommand** (Listing 9–6) is nothing but one big **switch** statement that relays control, in turn, to a set of even more specialized routines that handle each specific menu command. Notice that the identifiers used here to designate the various menu items (**New_Item**, **Open_Item**, and so on) are the same ones we saw earlier in Listing 9–1. These are constants defined by the program itself in the header file **WiniEdit Resources.h**. I associated them with the menu items when I built the menu onscreen with the Visual C++ interactive menu editor; the editor then incorporated them into the menu's resource description, as you can see by referring back to Listing 9–2.

Keyboard Shortcuts

As on the Macintosh, you can provide keyboard shortcuts to allow the user to invoke menu items from the keyboard rather than with the mouse. In fact, Windows allows two different kinds of keyboard shortcut: mnemonics and accelerators. Although they both serve the same basic purpose, the two have evolved separately, side by side, for different historical reasons. A good Windows program should support both.

Mnemonics

Right from the start, the Macintosh was designed with the mouse in mind. The mouse (or trackball or other equivalent pointing device) is an integral, indispensable part of the system: no Macintosh has ever been sold without one, or ever will be. In the IBM/DOS/Intel world, the mouse has always been regarded as an optional peripheral device, a gimmicky afterthought on what was essentially a keyboard- and character-based system. So Windows programs have always been expected to accommodate their mouseless users by providing a keyboard alternative for every mouse action.

In keeping with that principle, Windows defines a standard keyboard interface for choosing items from a menu:

- The Alt key toggles menu selection mode on or off. When first pressed, it enters menu mode and highlights the first item in the menu bar (normally a menu title); once in menu mode, pressing the Alt key escapes from it.
- The combination Alt-space displays the system menu.
- The left and right arrow keys cycle the highlight from item to item within the menu bar (including the system menu).
- The Enter key chooses the currently highlighted item, either activating a command or displaying a pop-up menu at the next level of the menu hierarchy.
- The up and down arrows cycle among the items within a pop-up menu.
- The Esc key backs up to the previous level of the menu hierarchy, or exits menu mode from the top level.

All of this standard behavior is implemented automatically by the default window procedure, so it takes no special effort on your part to support it. What you can do, however, is define *mnemonics* for your menus and menu items, allowing the user to choose them directly by typing a character from the keyboard while in menu mode, rather than laboriously cycling to them with the arrow keys. For example, if **F** is the mnemonic for the **File** menu and **O** is the mnemonic for the **Open...** item within that menu, then the user can invoke the **Open...** command by typing Alt to enter menu mode, then **F** for **File** and **O** for **Open...**.

You define a mnemonic for a menu item by including an ampersand character (**&**) in the item's text, either in your menu template resource or in the text string that you pass as a parameter to **AppendMenu** or **InsertMenu**. The character immediately following the ampersand will become the item's mnemonic. Windows will automatically underline the character when displaying that item on the menu, as a cue to the user, and will invoke the item when the user types the character in menu mode. In Listing 9–2, for instance, you'll notice that WiniEdit's menu template defines a pop-up menu named **&Edit** containing items named **&Undo**, **Cu&t**, **&Copy**, **&Paste**, **&Delete**, and **Select &All**. Windows will display the items with the indicated characters underlined (**Edit**, **Undo**, **Cut**, **Copy**, **Paste**, **Delete**, **Select All**), and will interpret these characters as mnemonics when typed in menu mode.

> In defining your mnemonics, be careful not to confuse your user by using the same character to stand for two different items on the same menu. It's your responsibility to make sure your mnemonics are unique and unambiguous—as WiniEdit does, for instance, by using **T** as the mnemonic for **Cut**, to distinguish it from **C** for **Copy**. (It's OK to use the same mnemonic character on two different menus, or at different levels of the menu hierarchy, but not twice on the same menu.) If you do make this mistake, Windows will utter no complaint; it will just cycle from one of the conflicting items to the next each time the user types the ambiguous mnemonic character.

Accelerators

The second form of keyboard shortcut, *accelerators*, are more like the ones we're used to on the Macintosh. Instead of entering a special "menu mode" with the Alt key, an accelerator is simply a modeless keystroke that the user can type at any time, such as Ctrl-**S** for **Save** or Ctrl-**P** for

Print. It's just a convenient alternative to using the mouse, not a substitute for the benefit of users who don't have a mouse in the first place.

You define your program's accelerators with a data structure called an *accelerator table.* Each entry in the table associates a keystroke with an item identifier, normally that of an item on one of your menus. You can either build the accelerator table from scratch with the Windows function **CreateAcceleratorTable** or (more commonly) define it as a resource and read it in at program initialization time with **LoadAccelerators**. Listing 9–7 shows the definition of WiniEdit's accelerator table, taken from the resource description file **WiniEdit.rc**. Like other resources that we've looked at, this one was built onscreen with the Visual C++ interactive resource editor and written out as a text-based resource description, then in turn compiled by the resource compiler into a resource of type **RT_ACCELERATOR** in the program's executable file.

Listing 9–7. WiniEdit accelerator table description

```
Accel_ID ACCELERATORS PRELOAD DISCARDABLE
BEGIN
    "A",            SelectAll_Item,         VIRTKEY, CONTROL
    "B",            Background_Item,        VIRTKEY, CONTROL
    "C",            Copy_Item,              VIRTKEY, CONTROL
    "D",            Default_Item,           VIRTKEY, CONTROL
    "F",            Format_Item,            VIRTKEY, CONTROL
    "N",            New_Item,               VIRTKEY, CONTROL
    "O",            Open_Item,              VIRTKEY, CONTROL
    "P",            Print_Item,             VIRTKEY, CONTROL
    "P",            Setup_Item,             VIRTKEY, CONTROL, ALT
    "Q",            Exit_Item,              VIRTKEY, CONTROL
    "R",            Revert_Item,            VIRTKEY, CONTROL
    "S",            Save_Item,              VIRTKEY, CONTROL
    "S",            SaveAs_Item,            VIRTKEY, CONTROL, ALT
    "V",            Paste_Item,             VIRTKEY, CONTROL
    VK_BACK,        Undo_Item,              VIRTKEY, ALT
    VK_DELETE,      Cut_Item,               VIRTKEY, SHIFT
    VK_F1,          Help_Item,              VIRTKEY
    VK_F2,          Cut_Item,               VIRTKEY
    VK_F3,          Copy_Item,              VIRTKEY
    VK_F4,          Paste_Item,             VIRTKEY
    VK_HELP,        Help_Item,              VIRTKEY, CONTROL
    VK_HELP,        Help_Item,              VIRTKEY, SHIFT, CONTROL
    VK_INSERT,      Copy_Item,              VIRTKEY, CONTROL
    VK_INSERT,      Paste_Item,             VIRTKEY, SHIFT
    "W",            Close_Item,             VIRTKEY, CONTROL
    "X",            Cut_Item,               VIRTKEY, CONTROL
    "Z",            Undo_Item,              VIRTKEY, CONTROL
END
```

Listing 9–8. Initialize accelerator table

```
VOID InitAccelerators (VOID)

   // Initialize accelerator table.

   {
      CHAR *resourceID;                                    // Resource ID in string form


      resourceID = MAKEINTRESOURCE(Accel_ID);              // Convert resource ID
      AccelTable = LoadAccelerators(ThisInstance, resourceID); // Load accelerator table

   }  /* end InitAccelerators */
```

The WiniEdit routine **`InitAccelerators`** (Listing 9–8), part of WiniEdit's one-time initialization sequence, reads in the accelerator table resource and stores its handle in a global variable, **`AccelTable`**. Later, as the program's message loop (Listing 9–9) retrieves messages from the message queue, it passes each one to the Windows function **`TranslateAccelerator`**, along with the global accelerator table handle. If the message is a keystroke (message type **`WM_KEYDOWN`**), **`TranslateAccelerator`** looks it up in the accelerator table. If it finds an accelerator for that keystroke, it generates an equivalent **`WM_COMMAND`** message and dispatches it directly to the program's window procedure (unless it's the identifier of a disabled or grayed menu item). The **`WM_COMMAND`** message carries a notification code of **`1`** to identify it as an accelerator command, along with the item identifier associated with the given keystroke in the accelerator table. (In Listing 9–7, for instance, if the

Listing 9–9. WiniEdit main program loop

```
VOID MainLoop (VOID)

   // Execute one pass of main program loop.

   {
      MSG   theMessage;                          // Next message to process
      BOOL  translated;                          // Was message translated as a keyboard accelerator?


      ContinueFlag = GetMessage(&theMessage, NULL, 0, 0);   // Get next message

      translated = TranslateAccelerator (TheWindow, AccelTable, &theMessage);
                                                 // Check for keyboard accelerator

      if ( !translated )                         // Was the message an accelerator?
         {
            TranslateMessage (&theMessage);      // Convert virtual keys to characters
            DispatchMessage  (&theMessage);      // Send message to window procedure

         }  /* end if ( !translated ) */

   }  /* end MainLoop */
```

keystroke were Ctrl-**A**, the corresponding item identifier would be **SelectAll_Item**.) **TranslateAccelerator** returns a boolean result to let the calling program know whether the incoming message was translated; if the result is **TRUE**, WiniEdit's message loop skips processing the raw keyboard message, knowing that an equivalent **WM_COMMAND** message has been dispatched in its place.

Although they resemble the keyboard shortcuts on the Macintosh, Windows accelerators are actually more flexible in a number of ways. For one thing, whereas Macintosh keyboard shortcuts are limited to combinations involving the Command key, Windows lets you define an accelerator for any keystroke you like. An accelerator can use any combination of the modifier keys Ctrl, Alt, and Shift, or it can use no modifiers at all. The WiniEdit accelerator table in Listing 9–7, for instance, defines the unmodified function keys **F1** to **F4** as accelerators for the **Help**, **Cut**, **Copy**, and **Paste** commands, respectively. In an application that doesn't require direct text entry from the keyboard, you could even use plain, unmodified letter keys as accelerators: **S** for **Save** instead of Ctrl-**S**, for example. Notice also that, unlike the Macintosh Toolbox, Windows lets you define more than one accelerator for the same operation. WiniEdit, for instance, provides three different accelerators for the **Cut** command: Shift-Delete (the Windows standard), Ctrl-**X** (the Macintosh standard, with the Ctrl key substituting for the Macintosh Command key), and **F2** (also common on the Macintosh).

Table 9–3 lists some accelerators that are considered standard in the Windows/DOS world. They aren't particularly logical or easy to remember, but Windows users expect them to work, so you should make sure your program supports them. The *system accelerators* listed in Table 9–4, on the other hand, are implemented automatically by the default window procedure; you don't need to anything special to support them, just make sure your own accelerators don't conflict with them. The Macintosh convention of using the first four keys on the bottom row of the keyboard for the standard editing commands (Ctrl-**Z** for **Undo**, Ctrl-**X** for **Cut**, Ctrl-**C** for **Copy**, Ctrl-**V** for **Paste**) also appears to be gaining acceptance in Windows, so you should probably support them, too. Many Macintosh programs also use the first four function keys (**F1**–**F4**) as shortcuts for these same four editing commands. Unfortunately, as shown in Table 9–4, **F1** is used almost universally in Windows as a help key, so it isn't available for the **Undo** command; but you can still use **F2**, **F3**, and **F4** for **Cut**, **Copy**, and **Paste**, as we've already seen that WiniEdit does.

Finally, notice that Windows accelerators, unlike their Macintosh counterparts, are not directly tied to the program's menu structure. On the Macintosh, by definition, every Command-key shortcut stands for a menu item. You can't define an operation that's strictly keyboard-based and not also available by selecting from a menu with the mouse. (You could implement such an operation by hand, of course, but there's no way to do it directly through the Toolbox.) Windows doesn't have this restriction: the item identifiers in the accelerator table are simply integer constants that you define for yourself in your program's resource header file. Although it's common to use the same identifier for both an accelerator and a menu item, there's no law that

Table 9–3. Standard Windows accelerators

Keystroke	Command equivalent
Alt-Backspace	**Undo**
Shift-Delete	**Cut**
Ctrl-Insert	**Copy**
Shift-Insert	**Paste**

Table 9–4. System accelerators

Keystroke	Meaning
Alt-Esc	Switch to next application
Shift-Alt-Esc	Switch to previous application
Alt-Tab	Cycle to next application
Shift-Alt-Tab	Cycle to previous application
Ctrl-Esc	Open task bar's Start menu
Alt-space	Open main window's system menu
Alt-hyphen	Open subwindow's system menu (Multiple Document Interface)
Alt-**F4**	Close main window
Ctrl-**F4**	Close subwindow (Multiple Document Interface)
F13	Copy snapshot of screen to clipboard
Alt-**F13**	Copy snapshot of active window to clipboard
F1	Help

this must be so. All the accelerator table says is that a certain keystroke typed by the user should be converted into a **WM_COMMAND** message with a certain value for its item identifier; the interpretation of that identifier is strictly up to your window procedure. So if you want to define an item identifier that corresponds to an accelerator keystroke only and has no corresponding menu item, you're free to do so. Whether this is a desirable user-interface feature can be debated either way; but the option is available if you want it.

Unlike the Macintosh Toolbox, Windows does *not* automatically incorporate keyboard accelerators into a menu when displaying it on the screen. If you want your accelerators to appear on the menu, you have to include them explicitly as part of each item's text. You can see an example of this in Listing 9–2, where, for instance, the text of the **Cut** menu item is defined as

```
Cu&t\tCtrl+X
```

As we saw in the last section, the ampersand (**&**) introduces the item's Alt-key mnemonic; the ampersand itself will not appear on the menu, but the character following it (in this case, **t**) will be underlined as the mnemonic. The item's accelerator, **Ctrl+X**, will be displayed next to it on the menu; the **\t** sets it off with a tab character to space it out to the right edge of the menu. The complete menu item will thus look like this:

Cut Ctrl+X

Windows will automatically calculate the tab width so that all the menu's contents align neatly on the screen. Note that the accepted onscreen convention for combination keystrokes in Windows is to separate the modifier key from the printable character with a plus sign (**+**), not a hyphen as on the Macintosh (and as I've been using in this book): **Ctrl+X**, not **Ctrl-X**.

When you create an accelerator table in the normal way, by reading it in as a resource with **LoadAccelerators**, you don't need to worry about disposing of it: Windows will destroy it for you automatically when your program terminates. However, if you build it from scratch with **CreateAcceleratorTable**, you have to destroy it explicitly with

`DestroyAcceleratorTable` before exiting from your program, or it will live on in memory after you're gone, like the Cheshire cat's grin.

Menu Operations

Table 9–5 summarizes some common Windows functions relating to menus and accelerators. We've already discussed some of them in this chapter; you can learn about the rest in the *Win32 Programmer's Reference.*

Table 9–5. Common menu functions

Function	Mac counterpart	Purpose
`CreateMenu`	————	Create top-level menu
`CreatePopupMenu`	`NewMenu`	Create pop-up menu
`LoadMenu`	`GetNewMBar`, `GetMenu`	Load menu from template resource
`LoadMenuIndirect`	————	Create menu from template in memory
`DestroyMenu`	`DisposeMenu`	Destroy menu
`AppendMenu`	`AppendMenu`	Add item at end of menu
`InsertMenu`	`InsertMenu`, `InsMenuItem`	Add item anywhere in menu
`DeleteMenu`	`DeleteMenu`, `DelMenuItem`	Delete item and destroy associated menu, if any
`RemoveMenu`	`DeleteMenu`, `DelMenuItem`	Remove item without destroying associated menu, if any
`SetMenu`	`SetMenuBar`	Set window's menu bar
`GetMenu`	`GetMenuBar`	Get handle to window's menu bar
`GetSubMenu`	`GetMHandle`	Get handle to pop-up menu
`GetSystemMenu`	————	Get handle to system menu
`IsMenu`	————	Is handle a menu handle?
`GetMenuItemCount`	`CountMItems`	Get number of items in menu
`GetMenuItemID`	————	Get item identifier by position
`GetMenuString`	`GetItem`	Get text of menu item
`GetMenuState`	————	Get menu item attributes
`ModifyMenu`	`SetItem`, `EnableItem`, `DisableItem`, `CheckItem`, `SetItemIcon`	Change text, appearance, or attributes of menu item
`EnableMenuItem`	`EnableItem`, `DisableItem`	Enable, disable, or gray menu item
`CheckMenuItem`	`CheckItem`	Check menu item
`SetMenuItemBitmaps`	`SetItemMark`	Set bitmap representing check mark
`GetMenuCheckMarkDimensions`	————	Get bitmap dimensions for check mark
`DrawMenuBar`	`DrawMenuBar`	Redraw menu bar
`HiliteMenuItem`	`HighlightMenu`	Highlight menu item on screen
`TrackPopupMenu`	`MenuSelect`	Track in menu
`CreateAcceleratorTable`	————	Create accelerator table in memory
`LoadAccelerators`	————	Load accelerator table from resource
`DestroyAcceleratorTable`	————	Destroy accelerator table
`CopyAcceleratorTable`	————	Copy accelerator table
`TranslateAccelerator`	`MenuKey`	Translate accelerator to menu command

The Same Thing...

- The Macintosh user can issue commands to a program by choosing menu items with the mouse.
- A Macintosh menu item can either invoke a command or pop up a submenu.
- Macintosh menu items can be disabled, preventing them from responding to the mouse.
- Macintosh menu items can be marked with a check mark or other symbol to show that they are currently in effect.
- Macintosh menus and menu bars can be built from scratch or read in as resources.
- A Macintosh program can define keyboard shortcuts to allow the user to invoke menu items from the keyboard.

- The Windows user can issue commands to a program by choosing menu items with the mouse.
- A Windows menu item can either invoke a command or pop up a submenu.
- Windows menu items can be disabled or grayed, preventing them from responding to the mouse.
- Windows menu items can be marked with a check mark or other symbol to show that they are currently in effect.
- Windows menus and menu bars can be built from scratch or read in as resources.
- A Windows program can define keyboard mnemonics and accelerators to allow the user to invoke menu items from the keyboard.

...Only Different

- The Macintosh has one global menu bar at the top of the screen.
- The Macintosh menu bar is a separate entity from the menus it contains.
- Every item in the Macintosh menu bar is a menu title.
- Macintosh menus disappear when the user releases the mouse button.

- A Windows menu bar belongs to a single window.
- A Windows menu bar is simply a menu that’s displayed horizontally instead of vertically.
- The Windows menu bar can contain command items as well as menu titles.
- Windows menus remain visible until the user clicks the mouse button a second time.

- The Macintosh has an Apple menu containing desk accessories and other frequently used items.
- Macintosh menu items are identified by their sequential position within the menu.
- When the user clicks the mouse in the menu bar, a Macintosh program must explicitly call a Toolbox tracking routine, **`MenuSelect`**, to track the mouse.
- A Macintosh program obtains the menu item chosen by the user as a direct function result returned by **`MenuSelect`**.
- A Macintosh program must either enable and disable its menu items incrementally as conditions change, or explicitly update the state of its menus before calling **`MenuSelect`**.
- Macintosh keyboard shortcuts are defined as part of the menu structure itself.
- Macintosh keyboard shortcuts are always invoked by key combinations involving the Command key.
- A Macintosh keyboard shortcut must invoke an existing menu item.
- A Macintosh menu item can have at most one keyboard shortcut.

- Windows has a system menu containing standard commands for manipulating the window on the screen.
- Windows menu items are identified by an arbitrary ID number assigned by the program.
- When the user clicks the mouse in the menu bar, a Windows program can leave the job of tracking the mouse to the default window procedure.
- A Windows program obtains the menu item chosen by via a **`WM_COMMAND`** message sent through the normal message mechanism.
- A Windows program receives a message, **`WM_INITMENUPOPUP`**, telling it when to update the state of a menu.
- Windows accelerators are defined in a separate accelerator table that can be read in as an independent resource.
- Windows accelerators can be invoked by any key combination at all.
- A Windows accelerator can invoke any operation at all, whether it exists as an independent menu item or not.
- A Windows menu item can have any number of accelerators.

Appendix A

WiniEdit Code Listing

Following is the complete listing of WiniEdit, the Windows version of the example Macintosh program, MiniEdit, from my *Macintosh Revealed* series. Although the program uses C++-style comments and a few other syntactic conveniences, it makes no use of the object-oriented features of C++ and is essentially written in straight C.

Header File

Here is the program's header file, **`WiniEdit.h`**:

```
//
//                                        WiniEdit.h
//                                Example Windows application program
//                               S. Chernicoff         15 January 1995
//

//                        Global header file for WiniEdit example application program


//-----------------------------------------------------------------------------------------------------


// Global Constants and Variables


   const INT  ClassNameMax = 32;                    // Maximum length for program name
   CHAR       ProgName[ClassNameMax];               // Name of program

   HANDLE  ThisInstance;                            // Handle to this instance of program
   HANDLE  PrevInstance;                            // Handle to previous instance of program

   HWND    TheWindow;                               // Handle to main window
   HWND    TheEditor;                               // Handle to edit control

   HACCEL  AccelTable;                              // Handle to accelerator table

   const INT   TextMargin = 4;                      // Inset from window to text rectangle
   CHOOSEFONT  FontParams;                          // Parameters for font dialog
```

```
   LOGFONT     TextFormat;                          // Current text formatting characteristics
   HFONT       TheFont;                             // Handle to current font
   HFONT       DeviceFont;                          // Handle to standard font for device

   CHOOSECOLOR  ColorParams;                        // Parameters for color dialog
   COLORREF     UserColors[16];                     // Array of user's custom colors
   COLORREF     TextColor;                          // Color for displaying text
   COLORREF     BackgroundColor;                    // Color for window background
   HBRUSH       BackgroundBrush;                    // Brush for painting window background

   const INT  DocMax = 32767;                       // Maximum length of text document
   const INT  MsgMax =   256;                       // Maximum length of on-screen messages

   HANDLE  TheFile = NULL;                          // Handle to current file

   const INT  PathMax = MAX_PATH;                   // Maximum length for file path names
   CHAR       PathName[PathMax];                    // Full path name of current file
   CHAR       NewPath [PathMax];                    // Full path name of new file

   const INT  TitleMax = 64;                        // Maximum length for title strings
   CHAR       FileTitle  [TitleMax];                // Local file name of current file
   CHAR       NewTitle   [TitleMax];                // Local file name of new file
   CHAR       NoNameTitle[TitleMax];                // Default title for window with no associated file

   const INT  FileExtMax = 4;                       // Maximum length for file extension
   CHAR       FileExt[FileExtMax];                  // Default file extension

   const INT  FilterMax = 256;                      // Maximum length for file filter string
   CHAR       FilterString[FilterMax];              // Filter string for Open and Save dialogs

   OPENFILENAME FileParams;                         // Parameters for file dialogs

   BOOL  ContinueFlag = TRUE;                       // Not time to quit yet?
   BOOL  ErrorFlag    = FALSE;                      // I/O error flag

//-------------------------------------------------------------------------------------------------------



// Function Prototypes


INT CALLBACK WinMain (HINSTANCE instHandle, HINSTANCE prevInst, LPSTR commandLine, INT showMode);
      // Main function
   VOID PASCAL Initialize (HANDLE thisInstance, HANDLE prevInstance, LPSTR commandLine, INT showMode);
         // Do one-time-only initialization.
      VOID InitProgram (VOID);
            // Do global program initialization.
      VOID InitInstance (LPSTR commandLine, INT showMode);
            // Initialize this instance of program.
         VOID InitWindow (LPSTR commandLine, INT showMode);
               // Initialize main window.
         VOID InitFileParams (VOID);
               // Initialize parameters for file dialogs.
```

```
      VOID InitTextFormat (VOID);
            // Initialize text format.
      VOID InitColorParams (VOID);
            // Initialize parameters for color dialog.
      VOID InitAccelerators (VOID);
            // Initialize accelerator table.
VOID MainLoop (VOID);
      // Execute one pass of main program loop.
LONG CALLBACK DoMessage (HWND thisWindow, UINT msgCode, WPARAM wParam, LPARAM lParam);
      // Get and process one message.
   VOID DoCreate (HWND thisWindow, WPARAM wParam, LPARAM lParam);
         // Handle WM_CREATE message.
   VOID DoSize (HWND thisWindow, WPARAM wParam, LPARAM lParam);
         // Handle WM_SIZE message.
   LONG DoCtlColorEdit (HWND thisWindow, WPARAM wParam, LPARAM lParam);
         // Handle WM_CTLCOLOREDIT message.
   VOID DoSysColorChange (HWND thisWindow, WPARAM wParam, LPARAM lParam);
         // Handle WM_SYSCOLORCHANGE message.
   VOID DoSetFocus (HWND thisWindow, WPARAM wParam, LPARAM lParam);
         // Handle WM_SETFOCUS message.
   VOID DoInitMenuPopup (HWND thisWindow, WPARAM wParam, LPARAM lParam);
         // Handle WM_INITMENUPOPUP message.
      VOID FixSystemMenu (VOID);
         // Enable/disable system menu commands.
      VOID FixFileMenu (HMENU theMenu);
         // Enable/disable File menu commands.
      VOID FixEditMenu (HMENU theMenu);
         // Enable/disable Edit menu commands.
      VOID FixFormatMenu (HMENU theMenu);
         // Enable/disable Format menu commands.
      VOID FixHelpMenu (HMENU theMenu);
         // Enable/disable Help menu commands.
   VOID DoCommand (HWND thisWindow, WPARAM wParam, LPARAM lParam);
         // Handle WM_COMMAND message.
      VOID DoMenuCommand (UINT itemID);
            // Handle menu command.
         VOID DoNew (VOID);
               // Handle New command.
         VOID DoOpen (VOID);
               // Handle Open... command.
            VOID ReadDoc (LPSTR newPath);
                  // Read document into window.
         VOID DoClose (VOID);
               // Handle Close command.
            BOOL CloseDoc (VOID);
                  // Close document displayed in window.
         VOID DoSave (VOID);
               // Handle Save command.
         VOID DoSaveAs (VOID);
               // Handle Save As... command.
            VOID WriteDoc (VOID);
                  // Write window contents to a file.
         VOID DoRevert (VOID);
               // Handle Revert to Saved... command.
```

```
        VOID DoSetup (VOID);
              // Handle Page Setup... command.
        VOID DoPrint (VOID);
              // Handle Print... command.
        VOID DoExit (VOID);
              // Handle Exit command.
        VOID DoUndo (VOID);
              // Handle Undo command.
        VOID DoCut (VOID);
              // Handle Cut command.
        VOID DoCopy (VOID);
              // Handle Copy command.
        VOID DoPaste (VOID);
              // Handle Paste command.
        VOID DoDelete (VOID);
              // Handle Delete command.
        VOID DoSelectAll (VOID);
              // Handle Select All command.
        VOID DoFormat (VOID);
              // Handle Format... command.
        VOID DoDefaultFormat (VOID);
              // Handle Default Format command.
           VOID SetFormat (VOID);
                 // Set new text format.
        VOID DoBackground (VOID);
              // Handle Background Color... command.
           VOID SetBackground (VOID);
                 // Set new background color.
        VOID DoHelp (VOID);
              // Handle Help command.
        VOID DoAbout (VOID);
              // Handle About WiniEdit... command.
           BOOL CALLBACK AboutProc (HWND thisWindow, UINT msgCode, WPARAM wParam, LPARAM lParam);
                 // Dialog procedure for About WiniEdit... dialog.
     VOID DoNotification (UINT itemID, HWND theControl, UINT notifyCode);
           // Handle control notification.
   BOOL DoQuery (HWND thisWindow, UINT wParam, LONG lParam);
         // Handle WM_QUERYENDSESSION message.
   VOID DoDestroy (HWND thisWindow, UINT wParam, LONG lParam);
         // Handle WM_DESTROY message.
VOID Finalize (VOID);
      // Do one-time-only finalization.
   INT ShowUserMessage (UINT msgID, UINT iconStyle, UINT buttonStyle, LPSTR msgTitle, LPSTR argString);
         // Display user message on screen.
   VOID IOCheck (VOID);
         // Check for I/O error.
```

Code File

This is the main code file, **`WiniEdit.cpp`**, containing the C++ source code for the WiniEdit program:

```
//
//                                          WiniEdit
//                                  Example Windows application program
//                                 S. Chernicoff         15 January 1995
//

//                      Sample program to display and manipulate a window on the screen


   #include <windows.h>
   #include "WiniEdit.h"
   #include "WiniEdit Resources.h"

//---------------------------------------------------------------------------------------------------------

INT CALLBACK WinMain (HINSTANCE instHandle, HINSTANCE prevHandle, LPSTR commandLine, INT showMode)

   // Main function

   {

      Initialize (instHandle, prevHandle, commandLine, showMode); // Do one-time-only initialization


      do
         MainLoop ();                              // Execute one pass of main loop
      while ( ContinueFlag );                      // Continue until time to quit


      Finalize ();                                 // Do one-time-only finalization


      return NO_ERROR;                             // Signal successful completion


   }  /* end WinMain */

//---------------------------------------------------------------------------------------------------------

VOID PASCAL Initialize (HANDLE instHandle, HANDLE prevHandle, LPSTR commandLine, INT showMode)

   // Do one-time-only initialization.

   {
      ThisInstance = instHandle;                   // Save global instance handle
```

```
      PrevInstance = prevHandle;                    // Save previous instance handle

      InitProgram ();                               // Do global program initialization
      InitInstance (commandLine, showMode);         // Initialize this instance of program

   }  /* end Initialize */

//-----------------------------------------------------------------------------------------------------

VOID InitProgram (VOID)

   // Do global program initialization.

   {
      WNDCLASS  windowClass;                                // Window class
      LPSTR     resourceID;                                 // Resource ID in string form
      HICON     progIcon;                                   // Program's screen icon
      HCURSOR   arrowCursor;                                // Default cursor
      HBRUSH    bkBrush;                                    // Brush for painting window background


      if ( PrevInstance == NULL )                           // Is this the first instance of program?
         {
            LoadString (ThisInstance, ProgName_Str,         // Get program name from resource
                        ProgName, ClassNameMax);
            windowClass.lpszClassName = ProgName;           // Use as class name

            windowClass.lpfnWndProc = WNDPROC(DoMessage);   // Set window procedure
            windowClass.hInstance   = ThisInstance;         // Current instance is the owner

            windowClass.style = CS_HREDRAW | CS_VREDRAW;    // Redraw on horizontal or vertical resize

            resourceID = MAKEINTRESOURCE(Main_Menu);        // Convert resource ID
            windowClass.lpszMenuName = resourceID;          // Set menu

            resourceID = MAKEINTRESOURCE(ProgIcon_ID);      // Convert resource ID
            progIcon   = LoadIcon(ThisInstance, resourceID);// Load icon
            windowClass.hIcon = progIcon;                   // Set icon

            arrowCursor = LoadCursor(NULL, IDC_ARROW);      // Load standard arrow cursor
            windowClass.hCursor = arrowCursor;              // Set cursor

            bkBrush = HBRUSH(COLOR_WINDOW + 1);             // Create brush for system background color
            windowClass.hbrBackground = bkBrush;            // Set window background brush

            windowClass.cbClsExtra = 0;                     // No extra class data
            windowClass.cbWndExtra = 0;                     // No extra window data

            RegisterClass (&windowClass);                   // Register the class

         }  /* end if ( PrevInstance == NULL ) */

   }  /* end InitProgram */
```

```
//-------------------------------------------------------------------------------------------------------

VOID InitInstance (LPSTR commandLine, INT showMode)

   // Initialize this instance of program.

   {
      InitWindow (commandLine, showMode);            // Initialize main window

      InitFileParams  ();                            // Initialize parameters for file dialogs
      InitTextFormat  ();                            // Initialize text format
      InitColorParams ();                            // Initialize parameters for color dialog

      InitAccelerators ();                           // Initialize accelerator table

   }  /* end InitInstance */

//-------------------------------------------------------------------------------------------------------

VOID InitWindow (LPSTR commandLine, INT showMode)

   // Initialize main window.

   {
      DWORD  windowStyle;                            // Style options for main window


      LoadString (ThisInstance, NoTitle_Str, NoNameTitle, TitleMax); // Get default title from resource


      windowStyle = WS_OVERLAPPEDWINDOW |            // Use standard window style
                    WS_CLIPCHILDREN;                 // Don't overwrite children

      TheWindow = CreateWindow (ProgName,            // Use program name for class name
                                NoNameTitle,         // Use default title for window with no associated file
                                windowStyle,         // Standard window with children clipped out
                                CW_USEDEFAULT,       // Let Windows choose initial x
                                CW_USEDEFAULT,       //   and y position
                                CW_USEDEFAULT,       // Let Windows choose initial width
                                CW_USEDEFAULT,       //   and height
                                NULL,                // No parent window
                                NULL,                // Use menu from window class
                                ThisInstance,        // Current program instance is the owner
                                NULL);               // No special creation parameters


      if ( commandLine[0] != '\0' )                  // Is there an initial file to open?
         ReadDoc (commandLine);                      // Read it into the window


      ShowWindow   (TheWindow, showMode);            // Display window on screen
      UpdateWindow (TheWindow);                      // Force update of client area
```

```
   }  /* end InitWindow */

//-------------------------------------------------------------------------------------------------------------

VOID InitFileParams (VOID)

   // Initialize parameters for file dialogs.

   {
      INT   filterLength;                          // Length of filter string
      INT   charIndex;                             // Index into filter string
      CHAR  dummyChar;                             // Character used as placeholder in filter string


      FileParams.lStructSize = sizeof(OPENFILENAME);  // Size of data structure

      FileParams.hwndOwner = TheWindow;            // Owning window
      FileParams.hInstance = ThisInstance;         // Owning instance of program

      FileParams.lpstrTitle = NULL;                // Use standard title for dialog window
      FileParams.Flags      = OFN_CREATEPROMPT |   // Prompt if file doesn't exist on open
                              OFN_OVERWRITEPROMPT;  //    or already exists on save

      filterLength = LoadString(ThisInstance, FileFilter_Str,           // Get filter string
                                FilterString, FilterMax);
      dummyChar = FilterString[filterLength - 1];                       // Last character is placeholder
      for ( (charIndex = 0); (charIndex < filterLength); (charIndex++) )  // Step through filter string
         if ( FilterString[charIndex] == dummyChar )                    // Find placeholders and
            FilterString[charIndex] = '\0';                             //    replace them with nulls

      FileParams.lpstrFilter  = FilterString;      // Point to filter string
      FileParams.nFilterIndex = 1;                 // Select first filter in string

      FileParams.lpstrInitialDir = NULL;           // Start with current directory

      LoadString (ThisInstance, FileExt_Str,       // Get default extension from resource
                  FileExt, FileExtMax);
      FileParams.lpstrDefExt = FileExt;            // Set default extension

      NewPath[0] = '\0';                           // Start with null path name
      FileParams.lpstrFile = NewPath;              // Pointer to path buffer
      FileParams.nMaxFile  = PathMax;              // Length of path buffer

      NewTitle[0] = '\0';                          // Start with null file title
      FileParams.lpstrFileTitle = NewTitle;        // Pointer to title buffer
      FileParams.nMaxFileTitle  = TitleMax;        // Length of title buffer

      FileParams.nFileOffset    = 0;               // File and extension offsets
      FileParams.nFileExtension = 0;               //    will be returned at time of call

      FileParams.lpTemplateName    = NULL;         // No custom dialog template
      FileParams.lpfnHook          = NULL;         // No custom hook function
      FileParams.lCustData         = 0;            // Data for custom hook not needed
      FileParams.lpstrCustomFilter = NULL;         // No custom filter buffer
```

```
      FileParams.nMaxCustFilter    = 0;              // Bufferlength is zero

   }  /* end InitFileParams */

//-------------------------------------------------------------------------------------------------------

VOID InitTextFormat (VOID)

   // Initialize text format.

   {

      DeviceFont = GetStockObject(DEVICE_DEFAULT_FONT);        // Get standard device font
      GetObject (DeviceFont, sizeof(TextFormat), &TextFormat); // Initialize logical font characteristics
      TheFont = CreateFontIndirect(&TextFormat);               // Save as current font


      FontParams.lStructSize = sizeof(CHOOSEFONT);             // Size of data structure

      FontParams.hwndOwner = TheWindow;                        // Owning window
      FontParams.hInstance = ThisInstance;                     // Owning instance of program

      FontParams.Flags = CF_INITTOLOGFONTSTRUCT |              // Initialize dialog from logical font
                         CF_SCREENFONTS |                      // List screen fonts only
                         CF_EFFECTS;                           // Enable effects

      FontParams.lpLogFont = &TextFormat;                      // Use for initial font
      FontParams.nFontType = 0;                                // Font type will be returned at time of call

      FontParams.iPointSize = 0;                               // Point size will be returned at time of call
      FontParams.nSizeMin   = 0;                               // No minimum point size
      FontParams.nSizeMax   = 0;                               // No maximum point size

      TextColor = 0;                                           // Default to system text color
      FontParams.rgbColors = GetSysColor(COLOR_WINDOWTEXT);    // Start with system text color

      FontParams.hDC            = NULL;                        // No printer context
      FontParams.lpszStyle      = NULL;                        // No style buffer
      FontParams.lpTemplateName = NULL;                        // No custom dialog template
      FontParams.lpfnHook       = NULL;                        // No custom hook function
      FontParams.lCustData      = 0;                           // Data for custom hook not needed


   }  /* end InitTextFormat */

//-------------------------------------------------------------------------------------------------------

VOID InitColorParams (VOID)

   // Initialize parameters for color dialog.

   {
      ColorParams.lStructSize = sizeof(CHOOSECOLOR);  // Size of data structure
```

```
      ColorParams.hwndOwner = TheWindow;             // Owning window
      ColorParams.hInstance = ThisInstance;          // Owning instance of program

      ColorParams.Flags = CC_RGBINIT;                // Use initial color setting

      BackgroundColor = 0;                                 // Default to system background color
      ColorParams.rgbResult    = GetSysColor(COLOR_WINDOW); // Start with system background color
      ColorParams.lpCustColors = UserColors;               // Pointer to custom color array

      ColorParams.lpTemplateName = NULL;             // No custom dialog template
      ColorParams.lpfnHook       = NULL;             // No custom hook function
      ColorParams.lCustData      = 0;                // Data for custom hook not needed

   }  /* end InitColorParams */

//-----------------------------------------------------------------------------------------------------------

VOID InitAccelerators (VOID)

   // Initialize accelerator table.

   {
      LPSTR resourceID;                                    // Resource ID in string form


      resourceID = MAKEINTRESOURCE(Accel_ID);              // Convert resource ID
      AccelTable = LoadAccelerators(ThisInstance, resourceID); // Load accelerator table

   }  /* end InitAccelerators */

//-----------------------------------------------------------------------------------------------------------

VOID MainLoop (VOID)

   // Execute one pass of main program loop.

   {
      MSG   theMessage;                              // Next message to process
      BOOL  translated;                              // Was message translated as a keyboard accelerator?


      ContinueFlag = GetMessage(&theMessage, NULL, 0, 0);   // Get next message

      translated = TranslateAccelerator (TheWindow, AccelTable, &theMessage); // Check for keyboard accelerator

      if ( !translated )                             // Was the message an accelerator?
         {
            TranslateMessage (&theMessage);          // Convert virtual keys to characters
            DispatchMessage  (&theMessage);          // Send message to window procedure

         }  /* end if ( !translated ) */

   }  /* end MainLoop */
```

```
//-------------------------------------------------------------------------------------------------------

LONG CALLBACK DoMessage (HWND thisWindow, UINT msgCode, WPARAM wParam, LPARAM lParam)

   // Get and process one message.

   {
      LONG  result = 0;                                    // Function result


      ErrorFlag = FALSE;                                   // Clear I/O error flag


      switch ( msgCode )                                   // Dispatch on message code
         {
            case WM_CREATE:
               DoCreate (thisWindow, wParam, lParam);          // Handle WM_CREATE message
               break;

            case WM_SIZE:
               DoSize (thisWindow, wParam, lParam);            // Handle WM_SIZE message
               break;

            case WM_CTLCOLOREDIT:
               result = DoCtlColorEdit (thisWindow, wParam, lParam);    // Handle WM_CTLCOLOREDIT message
               break;

            case WM_SYSCOLORCHANGE:
               DoSysColorChange (thisWindow, wParam, lParam);  // Handle WM_SYSCOLORCHANGE message
               break;

            case WM_SETFOCUS:
               DoSetFocus (thisWindow, wParam, lParam);        // Handle WM_SETFOCUS message
               break;

            case WM_INITMENUPOPUP:
               DoInitMenuPopup (thisWindow, wParam, lParam);   // Handle WM_INITMENUPOPUP message
               break;

            case WM_COMMAND:
               DoCommand (thisWindow, wParam, lParam);         // Handle WM_COMMAND message
               break;

            case WM_QUERYENDSESSION:
               result = DoQuery (thisWindow, wParam, lParam);  // Handle WM_QUERYENDSESSION message
               break;

            case WM_CLOSE:
               DoClose ();                                     // Handle WM_CLOSE message
               break;

            case WM_DESTROY:
               DoDestroy (thisWindow, wParam, lParam);         // Handle WM_DESTROY message
               break;
```

```
         default:
            result = DefWindowProc (thisWindow, msgCode,    // Pass message to Windows
                                    wParam, lParam);        //    for default processing
            break;

      }  /* end switch ( msgCode ) */


   return result;


}  /* end DoMessage */

//---------------------------------------------------------------------------------------------------------

VOID DoCreate (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_CREATE message.

   {
      DWORD     editStyle;                          // Style options for edit control
      COLORREF  bkColor;                            // Color for window background


      editStyle = WS_CHILD |                        // Child window
                  WS_VISIBLE |                      //    visible on screen
                  WS_VSCROLL |                      //    vertical scroll bar
                  ES_AUTOVSCROLL |                  //    vertical autoscroll
                  ES_MULTILINE |                    //    multiple lines of text
                  ES_LEFT;                          //    flush-left alignment

      TheEditor = CreateWindow ("EDIT",             // Standard edit control
                                NULL,               // No title
                                editStyle,          // Style options as above
                                0, 0, 0, 0,         // Position and size will be set later
                                thisWindow,         // Main window is the parent
                                HMENU(Edit_Control),// Child identifier for notification messages
                                ThisInstance,       // Current program instance is the owner
                                NULL);              // No special creation parameters


      lstrcpy (FileTitle, NoNameTitle);             // Set default file title


      SendMessage (TheEditor, WM_SETFONT,           // Set to current text font
                   WPARAM(TheFont), 0);

      if ( BackgroundColor != 0 )                   // Is there an explicit background color?
         bkColor = BackgroundColor;                 // Use it
      else
         bkColor = GetSysColor(COLOR_WINDOW);       // Otherwise use system background color
      BackgroundBrush = CreateSolidBrush(bkColor);  // Create background brush
```

```
   }  /* end DoCreate */

//-----------------------------------------------------------------------------------------------------

VOID DoSize (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_SIZE message.

   {
      INT     newWidth;                                  // New width of client area
      INT     newHeight;                                    // New height of client area
      RECT    textRect;                                  // Formatting rectangle for wrapping text
      LPARAM  rectParam;                                 // Pointer to rectangle as long-word parameter


      newWidth  = LOWORD(lParam);                        // Extract new dimensions
      newHeight = HIWORD(lParam);                        //    from message parameter

      MoveWindow (TheEditor,                             // Resize edit control to fit
                  0, 0,
                  newWidth, newHeight,
                  TRUE);

      rectParam = LPARAM(&textRect);                     // Convert to long integer
      SendMessage (TheEditor, EM_GETRECT, 0, rectParam); // Get formatting rectangle
      InflateRect (&textRect, -TextMargin, -TextMargin); // Inset by text margin
      SendMessage (TheEditor, EM_SETRECT, 0, rectParam); // Set new rectangle

   }  /* end DoSize */

//-----------------------------------------------------------------------------------------------------

LONG DoCtlColorEdit (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_CTLCOLOREDIT message.

   {
      HDC       theContext;                          // Handle to device context
      COLORREF  txColor;                             // Color for displaying text
      COLORREF  bkColor;                             // Color for window background


      theContext = HDC(wParam);                      // Get device context from parameter


      if ( TextColor != 0 )                          // Is there an explicit text color?
         txColor = TextColor;                        // Use it
      else
         txColor = GetSysColor(COLOR_WINDOWTEXT);    // Otherwise use system text color

      if ( BackgroundColor != 0 )                    // Is there an explicit background color?
         bkColor = BackgroundColor;                  // Use it
      else
```

```
         bkColor = GetSysColor(COLOR_WINDOW);         // Otherwise use system background color


      SetTextColor (theContext, txColor);             // Set text color
      SetBkColor   (theContext, bkColor);             // Set background color


      return LONG(BackgroundBrush);                   // Return window's background brush


   }  /* end DoCtlColorEdit */

//-------------------------------------------------------------------------------------------------------

VOID DoSysColorChange (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_SYSCOLORCHANGE message.

   {
      COLORREF  txColor;                              // New system text color
      COLORREF  bkColor;                              // New system background color


      txColor = GetSysColor(COLOR_WINDOWTEXT);        // Get new text color
      bkColor = GetSysColor(COLOR_WINDOW);            // Get new background color


      if ( TextColor == 0 )                           // Is text defaulted to system color?
         FontParams.rgbColors = txColor;              // Set to new system text color
      else if ( txColor == TextColor )                // Does new color match existing setting?
         TextColor = 0;                               // Default back to system color


      if ( BackgroundColor == 0 )                     // Is background defaulted to system color?
         {
            ColorParams.rgbResult = bkColor;          // Set to new system background color

            DeleteObject (BackgroundBrush);              // Destroy old background brush
            BackgroundBrush = CreateSolidBrush(bkColor); // Create new brush

         }  /* end if ( BackgroundColor == 0 ) */
      else if ( bkColor == BackgroundColor )          // Does new color match existing setting?
         BackgroundColor = 0;                         // Default back to system color


      InvalidateRect (TheWindow, NULL, TRUE);         // Force repaint


   }  /* end DoSysColorChange */

//-------------------------------------------------------------------------------------------------------

VOID DoSetFocus (HWND thisWindow, WPARAM wParam, LPARAM lParam)
```

```
   // Handle WM_SETFOCUS message.

   {
      SetFocus (TheEditor);                        // Pass focus to the edit control

   }  /* end DoSetFocus */

//-----------------------------------------------------------------------------------------------------------

VOID DoInitMenuPopup (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_INITMENUPOPUP message.

   {
      HMENU  theMenu   = HMENU(wParam);            // Handle to menu to be adjusted
      UINT   menuIndex = LOWORD(lParam);           // Relative position of menu in menu bar
      BOOL   isSystem  = HIWORD(lParam);           // Is it the system menu?


      if ( isSystem )                              // Is it the system menu?
         FixSystemMenu ();                         // Enable/disable system menu commands
      else
         switch ( menuIndex )
            {
               case File_Menu:
                  FixFileMenu (theMenu);           // Enable/disable File menu commands
                  break;

               case Edit_Menu:
                  FixEditMenu (theMenu);           // Enable/disable Edit menu commands
                  break;

               case Format_Menu:
                  FixFormatMenu (theMenu);         // Enable/disable Format menu commands
                  break;

               case Help_Menu:
                  FixHelpMenu (theMenu);           // Enable/disable Help menu commands
                  break;

            }  /* end switch ( menuIndex ) */

   }  /* end DoInitMenuPopup */

//-----------------------------------------------------------------------------------------------------------

VOID FixSystemMenu (VOID)

   // Enable/disable system menu commands.

   {
      /* Nothing to do */

   }  /* end FixSystemMenu */
```

```
//-----------------------------------------------------------------------------------------------------------

VOID FixFileMenu (HMENU theMenu)

   // Enable/disable File menu commands.

   {
      BOOL  dirty;                                  // Have window's contents been changed since last save?


      EnableMenuItem (theMenu, New_Item,    MF_ENABLED);          // These commands are always available
      EnableMenuItem (theMenu, Open_Item,   MF_ENABLED);
      EnableMenuItem (theMenu, Close_Item,  MF_ENABLED);
      EnableMenuItem (theMenu, SaveAs_Item, MF_ENABLED);
      EnableMenuItem (theMenu, Exit_Item,   MF_ENABLED);

      EnableMenuItem (theMenu, Setup_Item, MF_GRAYED);            // Printing not implemented
      EnableMenuItem (theMenu, Print_Item, MF_GRAYED);


      dirty = SendMessage(TheEditor, EM_GETMODIFY, 0, 0);         // Ask if text has been edited
      if ( dirty )                                                // Has it been?
         {
            EnableMenuItem (theMenu, Save_Item, MF_ENABLED);      // Enable Save command

            if ( TheFile != NULL )                                // Is window associated with a file?
               EnableMenuItem (theMenu, Revert_Item, MF_ENABLED); // Enable Revert to Saved... command
            else
               EnableMenuItem (theMenu, Revert_Item, MF_GRAYED);  // If no file, gray out Revert

         }  /* end if ( dirty ) */
      else
         {
            EnableMenuItem (theMenu, Save_Item,   MF_GRAYED);     // If text not dirty,
            EnableMenuItem (theMenu, Revert_Item, MF_GRAYED);     //    gray out Save and Revert

         }  /* end else */


   }  /* end FixFileMenu */

//-----------------------------------------------------------------------------------------------------------

VOID FixEditMenu (HMENU theMenu)

   // Enable/disable Edit menu commands.

   {
      BOOL  canUndo;                                // Can editor support Undo command?
      LONG  selStart;                               // Character position at start of selection
      LONG  selEnd;                                 // Character position at end of selection
      BOOL  canPaste;                               // Is there text on the clipboard?
      LONG  textLength;                             // Number of characters in document
```

```
   canUndo = SendMessage(TheEditor, EM_CANUNDO, 0, 0);      // Is Undo available?
   if ( canUndo )
      EnableMenuItem (theMenu, Undo_Item, MF_ENABLED);      // Enable Undo command
   else
      EnableMenuItem (theMenu, Undo_Item, MF_GRAYED);       // Gray out Undo command


   SendMessage ( TheEditor, EM_GETSEL,                      // Get selection range
                 WPARAM(&selStart), LPARAM(&selEnd) );
   if ( selStart != selEnd )                                // Is there a selection?
      {
         EnableMenuItem (theMenu, Cut_Item,    MF_ENABLED); // Enable Cut command
         EnableMenuItem (theMenu, Copy_Item,   MF_ENABLED); // Enable Copy command
         EnableMenuItem (theMenu, Delete_Item, MF_ENABLED); // Enable Delete command

      }  /* end if ( selStart == selEnd ) */
   else
      {
         EnableMenuItem (theMenu, Cut_Item,    MF_GRAYED);  // Gray out Cut command
         EnableMenuItem (theMenu, Copy_Item,   MF_GRAYED);  // Gray out Copy command
         EnableMenuItem (theMenu, Delete_Item, MF_GRAYED);  // Gray out Delete command

      }  /* end else */


   canPaste = IsClipboardFormatAvailable(CF_TEXT);          // Does clipboard contain text?
   if ( canPaste )
      EnableMenuItem (theMenu, Paste_Item, MF_ENABLED);     // Enable Paste command
   else
      EnableMenuItem (theMenu, Paste_Item, MF_GRAYED);      // Gray out Paste command


   textLength = GetWindowTextLength(TheEditor);             // Get length of text
   if ( textLength > 0 )                                    // Any text in document?
      EnableMenuItem (theMenu, SelectAll_Item, MF_ENABLED); // Enable Select All command
   else
      EnableMenuItem (theMenu, SelectAll_Item, MF_GRAYED);  // Gray out Select All command


}  /* end FixEditMenu */

//-------------------------------------------------------------------------------------------------------

VOID FixFormatMenu (HMENU theMenu)

   // Enable/disable Format menu commands.

   {
      EnableMenuItem (theMenu, Format_Item,     MF_ENABLED);   // Commands always enabled
      EnableMenuItem (theMenu, Default_Item,    MF_ENABLED);
      EnableMenuItem (theMenu, Background_Item, MF_ENABLED);
```

```
   }  /* end FixFormatMenu */

//-----------------------------------------------------------------------------------------------------------

VOID FixHelpMenu (HMENU theMenu)

   // Enable/disable Help menu commands.

   {
      EnableMenuItem (theMenu, Help_Item,  MF_GRAYED);   // Gray out Help... command
      EnableMenuItem (theMenu, About_Item, MF_ENABLED);  // Enable About WiniEdit... command

   }  /* end FixHelpMenu */

//-----------------------------------------------------------------------------------------------------------

VOID DoCommand (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_COMMAND message.

   {
      UINT  notifyCode;                          // Notification code from child control
      UINT  itemID;                              // Item ID of message originator
      HWND  theControl;                          // Handle to control sending notification


      notifyCode = HIWORD(wParam);               // Extract notification code
      itemID     = LOWORD(wParam);               // Extract item ID
      theControl = HWND(lParam);                 // Get control handle

      switch ( notifyCode )
         {
            case 0:
            case 1:
               DoMenuCommand (itemID);                          // Handle menu command
               break;

            default:
               DoNotification (itemID, theControl, notifyCode);   // Handle control notification
               break;

         }  /* end switch ( notifyCode ) */

   }  /* end DoCommand */

//-----------------------------------------------------------------------------------------------------------

VOID DoMenuCommand (UINT itemID)

   // Handle menu command.

   {
      switch ( itemID )
         {
```

```
/* File menu */

case New_Item:
   DoNew ();                        // Handle New command
   break;

case Open_Item:
   DoOpen ();                       // Handle Open... command
   break;

case Close_Item:
   DoClose ();                      // Handle Close command
   break;

case Save_Item:
   DoSave ();                       // Handle Save command
   break;

case SaveAs_Item:
   DoSaveAs ();                     // Handle Save As... command
   break;

case Revert_Item:
   DoRevert ();                     // Handle Revert to Saved... command
   break;

case Setup_Item:
   DoSetup ();                      // Handle Page Setup... command
   break;

case Print_Item:
   DoPrint ();                      // Handle Print... command
   break;

case Exit_Item:
   DoExit ();                       // Handle Exit command
   break;


/* Edit menu */

case Undo_Item:
   DoUndo ();                       // Handle Undo command
   break;

case Cut_Item:
   DoCut ();                        // Handle Cut command
   break;

case Copy_Item:
   DoCopy ();                       // Handle Copy command
   break;
```

```
            case Paste_Item:
               DoPaste ();                          // Handle Paste command
               break;

            case Delete_Item:
               DoDelete ();                         // Handle Delete command
               break;

            case SelectAll_Item:
               DoSelectAll ();                      // Handle Select All command
               break;


            /* Format menu */

            case Format_Item:
               DoFormat ();                         // Handle Text Format... command
               break;

            case Default_Item:
               DoDefaultFormat ();                  // Handle Default Format command
               break;

            case Background_Item:
               DoBackground ();                     // Handle Background Color... command
               break;


            /* Help menu */

            case Help_Item:
               DoHelp ();                           // Handle Help command
               break;

            case About_Item:
               DoAbout ();                          // Handle About WiniEdit... command
               break;



            default:
               MessageBeep (MB_OK);                 // Error: control should never reach this point
               break;


         }  /* end switch ( itemID ) */


   }  /* end DoMenuCommand */

//-----------------------------------------------------------------------------------------------------

VOID DoNew (VOID)
```

```
   // Handle New command.

   {
      BOOL  confirmed;                          // Did user confirm operation?


      confirmed = CloseDoc ();                  // Allow user to save document if necessary
      if ( confirmed )                          // Did user confirm?
         {
            TheFile     = NULL;                 // Clear current file
            PathName[0] = '\0';                 // Clear path name
            lstrcpy (FileTitle, NoNameTitle);   // Set default file title

            SetWindowText (TheWindow, NoNameTitle);   // Set window title to default

         }  /* end if ( confirmed ) */

   }  /* end DoNew */

//-----------------------------------------------------------------------------------------------------------

VOID DoOpen (VOID)

   // Handle Open... command.

   {
      BOOL    confirmed;                        // Did user confirm operation?


      confirmed = GetOpenFileName (&FileParams);      // Get file name from user

      if ( confirmed )                          // Did user confirm file selection?
         ReadDoc (NewPath);                     // Open file and read into window

   }  /* end DoOpen */

//-----------------------------------------------------------------------------------------------------------

VOID ReadDoc (LPSTR *newPath)

   // Read document into window.

   {
      BOOL    confirmed;                        // Did user confirm operation?
      HANDLE  newFile;                          // Handle to new file


      confirmed = CloseDoc ();                  // Allow user to save previous document if necessary
      if ( confirmed )                          // Did user confirm?
         {
            newFile = CreateFile (newPath,                      // Open new file
                                  GENERIC_READ | GENERIC_WRITE,   // Read-write access
                                  FILE_SHARE_READ,                // Share for reading only
                                  NULL,                           // No security attributes
```

```
                              OPEN_ALWAYS,                  // Open existing file or create a new one
                              FILE_ATTRIBUTE_NORMAL,        // No special attributes
                              NULL);                        // No template file

            if ( newFile == INVALID_HANDLE_VALUE )          // Did operation fail?
               IOCheck ();                                  // Handle error
            else
               SetLastError (NO_ERROR);                     // Clear possible "already exists" error

            if ( ErrorFlag )                                // Was there an error?
               {
                  TheFile      = NULL;                     // Window is left with no file:
                  PathName[0]  = '\0';                     //    clear global file info
                  FileTitle[0] = '\0';

                  SetWindowText (TheWindow, NoNameTitle);   // Set window title to default

               }  /* end if ( ErrorFlag ) */
            else
               {
                  TheFile = newFile;                        // Make new file current
                  lstrcpy (PathName,  newPath);             // Save path name
                  GetFileTitle (PathName, FileTitle, TitleMax);   // Extract local file name

                  SetWindowText (TheWindow, FileTitle);     // Local file name becomes window title
                  DoRevert ();                              // Read file into window

               }  /* end else */

         }  /* end if ( confirmed ) */


   }  /* end ReadDoc */

//-----------------------------------------------------------------------------------------------------

VOID DoClose (VOID)

   // Handle Close command.

   {
      BOOL  confirmed;                          // Did user confirm operation?


      confirmed = CloseDoc ();                  // Allow user to save document if necessary
      if ( confirmed )                          // Did user confirm?
         DestroyWindow (TheWindow);             // Destroy the window

   }  /* end DoClose */

//-----------------------------------------------------------------------------------------------------

BOOL CloseDoc (VOID)
```

```
// Close document displayed in window.

{
   BOOL  dirty;                                  // Have window's contents been changed since last save?
   INT   msgResult;                              // Result value returned by message box
   BOOL  confirmed;                              // Did user confirm operation?


   dirty = SendMessage(TheEditor, EM_GETMODIFY, 0, 0);   // Ask if text has been edited
   if ( dirty )
      {
         msgResult = ShowUserMessage (Save_Msg,          // Ask user whether to save window contents
                                      MB_ICONQUESTION,   // Question-mark icon
                                      MB_YESNOCANCEL,    // Yes, No, and Cancel buttons
                                      "???",             // Title for message box
                                      FileTitle);        // Merge file title into message

         switch ( msgResult )                     // Dispatch on message result
            {
               case IDYES:
                  DoSave ();                      // Save window contents to disk
                  confirmed = !ErrorFlag;         // Confirm if no error
                  break;

               case IDNO:
                  confirmed = TRUE;               // Confirm without saving
                  break;

               case IDCANCEL:
                  confirmed = FALSE;              // Cancel operation
                  break;

            }  /* end switch ( msgResult ) */

      }  /* end if ( dirty ) */

   else
      confirmed = TRUE;                           // Confirm if not dirty


   if ( confirmed )                               // Did user confirm operation?
      {
         if ( (TheFile != NULL) )                 // Is window associated with a file?
            {
               CloseHandle (TheFile);             // Close file
               IOCheck ();                        // Check for I/O error

            }  /* end if ( (TheFile != NULL ) */

         SetWindowText (TheEditor, "");           // Clear edit control's text

      }  /* end if ( confirmed ) */
```

```
      return confirmed;                              // Report confirmation or cancellation


   }  /* end CloseDoc */

//-----------------------------------------------------------------------------------------------------------

VOID DoSave (VOID)

   // Handle Save command.

   {
      if ( TheFile == NULL )                         // Is window associated with a file?
         DoSaveAs ();                                // Get file name from user
      else
         WriteDoc ();                                // Write to window's file

   }  /* end DoSave */

//-----------------------------------------------------------------------------------------------------------

VOID DoSaveAs (VOID)

   // Handle Save As... command.

   {
      BOOL    confirmed;                             // Did user confirm dialog?
      HANDLE  newFile;                               // Handle to new file


      lstrcpy (NewPath,  PathName);                  // Start with current file name
      lstrcpy (NewTitle, FileTitle);


      confirmed = GetSaveFileName (&FileParams);     // Get file name from user
      if ( !confirmed )                              // Did user cancel?
         {
            ErrorFlag = TRUE;                        // Force exit to main message loop
            return;                                  // Skip rest of operation

         }  /* end if ( !confirmed ) */


      if ( lstrcmp(PathName, NewPath) != 0 )                    // Changing to a different file?
         {

            if ( TheFile != NULL )                              // Was there a previous file?
               {
                  CloseHandle (TheFile);                        // Close old file
                  IOCheck ();                                   // Check for error
                  if (ErrorFlag) return;                        // On error, exit to main message loop

               }  /* end if ( TheFile != NULL ) */
```

```
         newFile = CreateFile (NewPath,                        // Open new file
                               GENERIC_READ | GENERIC_WRITE,   // Read-write access
                               FILE_SHARE_READ,                // Share for reading only
                               NULL,                           // No security attributes
                               CREATE_ALWAYS,                  // Overwrite existing file or create a new one
                               FILE_ATTRIBUTE_NORMAL,          // No special attributes
                               NULL);                          // No template file

         if ( newFile == INVALID_HANDLE_VALUE )                // Did operation fail?
            IOCheck ();                                        // Handle error
         else
            SetLastError (NO_ERROR);                           // Clear possible "already exists" error


         if ( ErrorFlag )                                      // Was there an error?
            {
               TheFile      = NULL;                           // Window is left with no file:
               PathName[0]  = '\0';                           //    clear global file info
               FileTitle[0] = '\0';

               SetWindowText (TheWindow, NoNameTitle);         // Set window title to default

               return;                                         // Exit to main message loop

            }  /* end if ( ErrorFlag ) */
         else
            {
               TheFile = newFile;                              // Make new file current
               lstrcpy (PathName,  NewPath);
               lstrcpy (FileTitle, NewTitle);

               SetWindowText (TheWindow, NewTitle);            // Local file name becomes window title

            }  /* end else */


      }  /* end if ( lstrcmp(PathName, NewPath) != 0 ) */


   WriteDoc ();                                                // Write window's contents to file


}  /* end DoSaveAs */


//-------------------------------------------------------------------------------------------------

VOID WriteDoc (VOID)

   // Write window contents to a file.

   {
      CHAR   textBuffer[DocMax];                    // Buffer to hold text of document
```

```
      INT    textLength;                              // Length of text in bytes
      LONG   textAddr;                                // Address of text buffer
      ULONG  bytesWritten;                            // Number of bytes written to file


      textLength = GetWindowTextLength(TheEditor);    // Get length of text
      textLength++;                                   // Adjust for terminating null character
      textAddr = LPARAM(textBuffer);                  // Convert to long parameter
      SendMessage (TheEditor, WM_GETTEXT, textLength, textAddr);  // Get text from edit control

      SetFilePointer (TheFile, 0, NULL, FILE_BEGIN);  // Reset file pointer to beginning
      IOCheck ();                                     // Check for error
      if (ErrorFlag) return;                          // On error, exit to main message loop

      WriteFile (TheFile, textBuffer, --textLength,   // Write text to file
                 &bytesWritten, NULL);
      IOCheck ();                                     // Check for error
      if (ErrorFlag) return;                          // On error, exit to main message loop

      SetEndOfFile (TheFile);                         // Set length of file
      IOCheck ();                                     // Check for error
      if (ErrorFlag) return;                          // On error, exit to main message loop

      FlushFileBuffers (TheFile);                     // Flush buffer to disk
      IOCheck ();                                     // Check for error
      if (ErrorFlag) return;                          // On error, exit to main message loop

      SendMessage(TheEditor, EM_SETMODIFY, FALSE, 0); // Mark text as clean

   }  /* end WriteDoc */

//-------------------------------------------------------------------------------------------------

VOID DoRevert (VOID)

   // Handle Revert to Saved... command.

   {
      BOOL   dirty;                                   // Have window's contents been changed since last save?
      INT    msgResult;                               // Result value returned by message box
      CHAR   textBuffer[DocMax];                      // Buffer to hold text of document
      LONG   textLength;                              // Length of text in bytes
      LONG   textAddr;                                // Address of text buffer
      ULONG  bytesRead;                               // Number of bytes read from file


      dirty = SendMessage(TheEditor, EM_GETMODIFY, 0, 0);   // Ask if text has been edited
      if ( dirty )                                          // Has it been?
         {
            msgResult = ShowUserMessage (Revert_Msg,        // Display revert message
                                         MB_ICONQUESTION,   // Question-mark icon
                                         MB_OKCANCEL,       // OK and Cancel buttons
                                         "???",             // Title for message box
                                         FileTitle);        // Merge file title into message
```

```
            if ( msgResult == IDCANCEL )                    // Did user cancel?
               return;                                      // Skip rest of operation

         }  /* end if ( dirty ) */

      textLength = GetFileSize(TheFile, NULL);          // Get length of file
      if ( textLength > DocMax )                        // File too long?
         {
            ShowUserMessage (TooLong_Msg,               // Display "file too long" message
                             MB_ICONEXCLAMATION,        // Exclamation-point icon
                             MB_OK,                     // OK button
                             NULL,                      // Use default title for message box
                             FileTitle);                // Merge file title into message

            CloseHandle (TheFile);                      // Close file
            ErrorFlag = TRUE;                           // Force exit

         }  /* end if ( textLength > DocMax ) */
      else
         IOCheck ();                                    // Check for I/O error
      if (ErrorFlag) return;                            // On error, exit to main message loop

      SetFilePointer (TheFile, 0, NULL, FILE_BEGIN);  // Move file pointer to beginning
      IOCheck ();                                       // Check for error
      if (ErrorFlag) return;                            // On error, exit to main message loop

      ReadFile (TheFile, textBuffer, textLength,        // Read text of file into block
                &bytesRead, NULL);
      textBuffer[textLength] = '\0';                    // Add terminating null character
      IOCheck ();                                       // Check for error
      if (ErrorFlag) return;                            // On error, exit to main message loop

      textAddr = LPARAM(textBuffer);                       // Convert to long parameter
      SendMessage (TheEditor, WM_SETTEXT, 0, textAddr);  // Copy file contents into edit control

   }  /* end DoRevert */

//-----------------------------------------------------------------------------------------------------------

VOID DoSetup (VOID)

   // Handle Page Setup... command.

   {
      MessageBeep (MB_OK);                              // Printing not implemented

   }  /* end DoSetup */

//-----------------------------------------------------------------------------------------------------------

VOID DoPrint (VOID)

   // Handle Print... command.
```

```
   {
      MessageBeep (MB_OK);                          // Printing not implemented

   }  /* end DoPrint */

//-----------------------------------------------------------------------------------------------------------

VOID DoExit (VOID)

   // Handle Exit command.

   {
      SendMessage (TheWindow, WM_CLOSE, 0, 0);      // Close the window

   }  /* end DoExit */

//-----------------------------------------------------------------------------------------------------------

VOID DoUndo (VOID)

   // Handle Undo command.

   {
      SendMessage (TheEditor, WM_UNDO, 0, 0);       // Relay operation to edit control

   }  /* end DoUndo */

//-----------------------------------------------------------------------------------------------------------

VOID DoCut (VOID)

   // Handle Cut command.

   {
      SendMessage (TheEditor, WM_CUT, 0, 0);        // Relay operation to edit control

   }  /* end DoCut */

//-----------------------------------------------------------------------------------------------------------

VOID DoCopy (VOID)

   // Handle Copy command.

   {
      SendMessage (TheEditor, WM_COPY, 0, 0);       // Relay operation to edit control

   }  /* end DoCopy */

//-----------------------------------------------------------------------------------------------------------

VOID DoPaste (VOID)

   // Handle Paste command.
```

```
   {
      SendMessage (TheEditor, WM_PASTE, 0, 0);        // Relay operation to edit control

   }  /* end DoPaste */

//-----------------------------------------------------------------------------------------------------------

VOID DoDelete (VOID)

   // Handle Delete command.

   {
      SendMessage (TheEditor, WM_CLEAR, 0, 0);        // Relay operation to edit control

   }  /* end DoDelete */

//-----------------------------------------------------------------------------------------------------------

VOID DoSelectAll (VOID)

   // Handle Select All command.

   {
      SendMessage (TheEditor, EM_SETSEL, 0, -1);      // Select entire document

   }  /* end DoSelectAll */

//-----------------------------------------------------------------------------------------------------------

VOID DoFormat (VOID)

   // Handle Format... command.

   {
      BOOL  confirmed;                                // Did user confirm dialog?


      confirmed = ChooseFont (&FontParams);           // Get new format from user
      if ( confirmed )                                // Did user confirm?
         SetFormat ();                                // Set new text format

   }  /* end DoFormat */

//-----------------------------------------------------------------------------------------------------------

VOID DoDefaultFormat (VOID)

   // Handle Default Format command.

   {
      INT       msgResult;                            // Result value returned by message box
      COLORREF  txColor;                              // System text color
      COLORREF  bkColor;                              // System background color
```

```
      msgResult = ShowUserMessage (DefaultFormat_Msg, // Display default format message
                                   MB_ICONQUESTION,   // Question-mark icon
                                   MB_OKCANCEL,       // OK and Cancel buttons
                                   "???",             // Title for message box
                                   FileTitle);        // Merge file title into message


      if ( msgResult != IDCANCEL )                    // Did user confirm?
         {
            GetObject (DeviceFont, sizeof(TextFormat), &TextFormat); // Get standard font characteristics

            txColor = GetSysColor(COLOR_WINDOWTEXT);  // Get system text color
            bkColor = GetSysColor(COLOR_WINDOW);      // Get system background color

            FontParams.rgbColors  = txColor;          // Set text to system color
            ColorParams.rgbResult = bkColor;          // Set background to system color

            SetFormat ();                             // Set new text format
            SetBackground ();                         // Set new background color

         }  /* end if ( msgResult != IDCANCEL ) */


   }  /* end DoDefaultFormat */

//-------------------------------------------------------------------------------------------------------

VOID SetFormat (VOID)

   // Set new text format.

   {
      RECT      textRect;                             // Formatting rectangle for wrapping text
      LPARAM    redrawFlag;                           // Redraw parameter for setting font
      COLORREF  newColor;                             // Requested text color
      COLORREF  sysColor;                             // System text color


      SendMessage (TheEditor, EM_GETRECT,             // Get formatting rectangle
                   0, LPARAM(&textRect));

         DeleteObject (TheFont);                         // Delete previous font
         TheFont    = CreateFontIndirect(&TextFormat);   // Create new font
         redrawFlag = MAKELPARAM(TRUE, 0);               // Set flag to redraw text
         SendMessage (TheEditor, WM_SETFONT,             // Set new font
                      WPARAM(TheFont), redrawFlag);

         newColor = FontParams.rgbColors;             // Get requested color
         sysColor = GetSysColor(COLOR_WINDOWTEXT);    // Get system text color

         if ( newColor == sysColor )                  // Did user choose system color?
            TextColor = 0;                            // Default back to system color
```

```
         else
            TextColor = newColor;                    // Save requested color

      SendMessage (TheEditor, EM_SETRECT,            // Restore formatting rectangle
                   0, LPARAM(&textRect));

   }  /* end SetFormat */

//-----------------------------------------------------------------------------------------------------

VOID DoBackground (VOID)

   // Handle Background Color... command.

   {
      BOOL  confirmed;                               // Did user confirm dialog?


      confirmed = ChooseColor (&ColorParams);        // Get new color from user
      if ( confirmed )                               // Did user confirm?
         SetBackground ();                           // Set new background color

   }  /* end DoBackground */

//-----------------------------------------------------------------------------------------------------

VOID SetBackground (VOID)

   // Set new background color.

   {
      COLORREF  newColor;                            // Requested background color
      COLORREF  sysColor;                            // System background color


      newColor = ColorParams.rgbResult;              // Get requested color
      sysColor = GetSysColor(COLOR_WINDOW);          // Get system background color

      if ( newColor == sysColor )                    // Did user choose system color?
         BackgroundColor = 0;                        // Default back to system color
      else
         BackgroundColor = newColor;                 // Save requested color

      DeleteObject (BackgroundBrush);                // Destroy old background brush
      BackgroundBrush = CreateSolidBrush(newColor);  // Create new brush

      InvalidateRect (TheEditor, NULL, TRUE);        // Force repaint

   }  /* end SetBackground */

//-----------------------------------------------------------------------------------------------------

VOID DoHelp (VOID)
```

```
   // Handle Help command.

   {
      MessageBeep (MB_OK);                          // Help not implemented

   }  /* end DoHelp */

//-----------------------------------------------------------------------------------------------------------

VOID DoAbout (VOID)

   // Handle About WiniEdit... command.

   {
      LPSTR  resourceID;                            // Resource ID in string form


      resourceID = MAKEINTRESOURCE(About_Dialog);    // Convert resource ID
      DialogBox ( ThisInstance, resourceID,         // Invoke dialog
                  TheWindow, DLGPROC(AboutProc) );

   }  /* end DoAbout */

//-----------------------------------------------------------------------------------------------------------

BOOL CALLBACK AboutProc (HWND theDialog, UINT msgCode, WPARAM wParam, LPARAM lParam)

   // Dialog procedure for About WiniEdit... dialog.

   {
      BOOL  result;                                 // Function result


      switch ( msgCode )                            // Dispatch on message code
         {
            case WM_INITDIALOG:
               result = TRUE;                       // Tell Windows to set the input focus
               break;

            case WM_COMMAND:
               switch ( wParam )                    // Dispatch on item ID
                  {
                     case IDOK:                     // Continue button, space bar, or Enter key
                     case IDCANCEL:                 // Esc key
                        EndDialog (theDialog, TRUE);// Dismiss the dialog
                        result = TRUE;              // Indicate message accepted
                        break;

                     default:
                        result = FALSE;             // Indicate message not accepted
                        break;

                  }  /* end switch ( wParam ) */
               break;
```

```
            default:
               result = FALSE;                        // Indicate message not accepted
               break;

         }  /* end switch ( msgCode ) */

      return result;

   }  /* end AboutProc */

//-----------------------------------------------------------------------------------------------------

VOID DoNotification (UINT itemID, HWND theControl, UINT notifyCode)

   // Handle control notification.

   {
      /* Nothing to do */

   }  /* end DoNotification */

//-----------------------------------------------------------------------------------------------------

BOOL DoQuery (HWND thisWindow, UINT wParam, LONG lParam)

   // Handle WM_QUERYENDSESSION message.

   {
      BOOL  confirmed;                                // Did user confirm operation?


      confirmed = CloseDoc ();                        // Allow user to save document if necessary
      return confirmed;                               // Report confirmation or cancellation

   }  /* end DoQuery */

//-----------------------------------------------------------------------------------------------------

VOID DoDestroy (HWND thisWindow, UINT wParam, LONG lParam)

   // Handle WM_DESTROY message.

   {
      DeleteObject (BackgroundBrush);                 // Destroy background brush

      PostQuitMessage (NO_ERROR);                     // Signal program completion

   }  /* end DoDestroy */

//-----------------------------------------------------------------------------------------------------

VOID Finalize (VOID)
```

```
   // Do one-time-only finalization.

   {
      /* Nothing to do */

   }  /* end Finalize */

//-----------------------------------------------------------------------------------------------------

INT ShowUserMessage (UINT msgID, UINT iconStyle, UINT buttonStyle, LPSTR msgTitle, LPSTR argString)

   // Display user message on screen.

   {
      CHAR     msgFormat[MsgMax];                  // Format for confirmation message
      CHAR     msgText  [MsgMax];                  // Text of confirmation message
      UINT     msgStyle;                           // Style options for message box
      INT      msgResult;                          // Result value returned by message box


      MessageBeep (MB_OK);                         // Beep for attention


      LoadString (ThisInstance, msgID,             // Get message text from resource
                  msgFormat, MsgMax );
      wsprintf   (msgText, msgFormat, argString);  // Merge argument string into message

      msgStyle = MB_APPLMODAL |                    // Application-modal message box
                 iconStyle |                       // Merge in icon style
                 buttonStyle |                     // Merge in button style
                 MB_SETFOREGROUND;                 // Bring to foreground

      msgResult = MessageBox (TheWindow, msgText,  // Display message box
                              msgTitle, msgStyle);


      return msgResult;                            // Pass result back to caller


   }  /* end ShowUserMessage */

//-----------------------------------------------------------------------------------------------------

VOID IOCheck (VOID)

   // Check for I/O error.

   {
      LONG    errorCode;                           // Code number identifying error
      CHAR    errorString[10];                     // Error code in string form


      errorCode = GetLastError();                  // Get error code
```

```
   if ( errorCode == NO_ERROR )                  // Was there an error?
      ErrorFlag = FALSE;                         // If not, just continue normal processing

   else
      {
         ultoa (errorCode, errorString, 10);     // Convert to string
         ShowUserMessage (IOError_Msg,           // Display generic I/O error message
                          MB_ICONEXCLAMATION,    // Exclamation-point icon
                          MB_OK,                 // OK button
                          NULL,                  // Use default title for message box
                          errorString);          // Merge error code into message

         ErrorFlag = TRUE;                       // Force exit to main message loop

      }  /* end else */

}  /* end IOCheck */
```

Resource Header File

The resource header file, **`WiniEdit Resources.h`**, defines the symbolic identification codes for the program's resources. This file was generated automatically by the Visual C++ onscreen resource editors, using the symbolic names and ID numbers I specified when building the program's resources with the onscreen resource editors. For better readability, I have done some editing on the raw output generated by the development software, such as grouping together resources of the same type and indenting definitions to reflect their hierarchical relationships.

```
//
//                                WiniEdit Resources.h
//                Resource header for example Windows application program
//                         S. Chernicoff         15 January 1995
//

//           Global resource header file for WiniEdit example application program


//---------------------------------------------------------------------------


// Icon

#define ProgIcon_ID                    1000


//---------------------------------------------------------------------------


// Menus

#define Main_Menu                      1000
```

```
#define File_Menu                   0
#define    New_Item                    1001
#define    Open_Item                   1002
#define    Close_Item                  1003
#define    Save_Item                   1004
#define    SaveAs_Item                 1005
#define    Revert_Item                 1006
#define    Setup_Item                  1007
#define    Print_Item                  1008
#define    Exit_Item                   1009

#define Edit_Menu                   1
#define    Undo_Item                   1101
#define    Cut_Item                    1102
#define    Copy_Item                   1103
#define    Paste_Item                  1104
#define    Delete_Item                 1105
#define    SelectAll_Item              1106

#define Format_Menu                 2
#define    Format_Item                 1201
#define    Default_Item                1202
#define    Background_Item             1203

#define Help_Menu                   3
#define    Help_Item                   1301
#define    About_Item                  1302


//---------------------------------------------------------------------------


// Accelerators

#define Accel_ID                       1000


//---------------------------------------------------------------------------


// Control ID

#define Edit_Control                   1000


//---------------------------------------------------------------------------


// Dialog

#define About_Dialog                   1000


//---------------------------------------------------------------------------
```

```
// Strings

#define ProgName_Str                    1001
#define NoTitle_Str                     1002
#define FileFilter_Str                  1003
#define FileExt_Str                     1004

#define Save_Msg                        2001
#define Revert_Msg                      2002
#define DefaultFormat_Msg               2003
#define WrongType_Msg                   2004
#define TooLong_Msg                     2005
#define OutOfMem_Msg                    2006
#define IOError_Msg                     2007


//------------------------------------------------------------------------------


// Next default values for new objects

#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        102
#define _APS_NEXT_COMMAND_VALUE         40004
#define _APS_NEXT_CONTROL_VALUE         1000
#define _APS_NEXT_SYMED_VALUE           102
#endif
#endif
```

Resource Description File

The resource description file, **`WiniEdit.rc`**, was generated automatically by the Visual C++ development software to define the resources I built onscreen with the interactive resource editors.

```
//Microsoft Visual C++ generated resource script.
//
#include "WiniEdit Resources.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS
```

```
#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "WiniEdit Resources.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

//////////////////////////////////////////////////////////////////////////////
#endif    // APSTUDIO_INVOKED


//////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

About_Dialog DIALOG DISCARDABLE  32, 32, 163, 96
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE
FONT 8, "MS Sans Serif"
BEGIN
    ICON            ProgIcon_ID,About_Dialog,8,8,18,20
    CTEXT           "WiniEdit 1.0",IDC_STATIC,61,12,41,8
    CTEXT           "Example Windows application",IDC_STATIC,32,32,98,8
    LTEXT           "S. Chernicoff",IDC_STATIC,8,52,44,8
    RTEXT           "15 January 1995",IDC_STATIC,100,52,55,8
    DEFPUSHBUTTON   "Continue",IDOK,60,68,43,20
END


//////////////////////////////////////////////////////////////////////////////
//
// Menu
//

Main_Menu MENU PRELOAD DISCARDABLE
BEGIN
```

```
    POPUP "&File"
    BEGIN
        MENUITEM "&New\tCtrl+N",                New_Item
        MENUITEM "&Open...\tCtrl+O",            Open_Item
        MENUITEM "&Close\tCtrl+W",              Close_Item
        MENUITEM SEPARATOR
        MENUITEM "&Save\tCtrl+S",               Save_Item
        MENUITEM "Save &As...\tCtrl+Alt+S",     SaveAs_Item
        MENUITEM "&Revert to Saved...\tCtrl+R", Revert_Item
        MENUITEM SEPARATOR
        MENUITEM "Page Set&up...\tCtrl+Alt+P",  Setup_Item
        MENUITEM "&Print...\tCtrl+P",           Print_Item
        MENUITEM SEPARATOR
        MENUITEM "E&xit\tCtrl+Q",               Exit_Item
    END
    POPUP "&Edit"
    BEGIN
        MENUITEM "&Undo\tCtrl+Z",               Undo_Item
        MENUITEM SEPARATOR
        MENUITEM "Cu&t\tCtrl+X",                Cut_Item
        MENUITEM "&Copy\tCtrl+C",               Copy_Item
        MENUITEM "&Paste\tCtrl+V",              Paste_Item
        MENUITEM "&Delete\tDelete",             Delete_Item
        MENUITEM SEPARATOR
        MENUITEM "Select &All\tCtrl+A",         SelectAll_Item
    END
    POPUP "&Format"
    BEGIN
        MENUITEM "Text &Format...\tCtrl+F",     Format_Item
        MENUITEM "&Default Format\tCtrl+D",     Default_Item
        MENUITEM SEPARATOR
        MENUITEM "&Background Color...\tCtrl+B", Background_Item
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&Help\tCtrl+?",               Help_Item
        MENUITEM SEPARATOR
        MENUITEM "&About WiniEdit...",          About_Item
    END
END


////////////////////////////////////////////////////////////////////////////
//
// Icon
//

ProgIcon_ID             ICON    DISCARDABLE     "WiniEdit.ico"

////////////////////////////////////////////////////////////////////////////
//
// Accelerator
//
```

```
Accel_ID ACCELERATORS PRELOAD DISCARDABLE
BEGIN
    "A",            SelectAll_Item,         VIRTKEY, CONTROL, NOINVERT
    "B",            Background_Item,        VIRTKEY, CONTROL, NOINVERT
    "C",            Copy_Item,              VIRTKEY, CONTROL, NOINVERT
    "D",            Default_Item,           VIRTKEY, CONTROL, NOINVERT
    "F",            Format_Item,            VIRTKEY, CONTROL, NOINVERT
    "N",            New_Item,               VIRTKEY, CONTROL, NOINVERT
    "O",            Open_Item,              VIRTKEY, CONTROL, NOINVERT
    "P",            Print_Item,             VIRTKEY, CONTROL, NOINVERT
    "P",            Setup_Item,             VIRTKEY, CONTROL, ALT, NOINVERT
    "Q",            Exit_Item,              VIRTKEY, CONTROL, NOINVERT
    "R",            Revert_Item,            VIRTKEY, CONTROL, NOINVERT
    "S",            Save_Item,              VIRTKEY, CONTROL, NOINVERT
    "S",            SaveAs_Item,            VIRTKEY, CONTROL, ALT, NOINVERT
    "V",            Paste_Item,             VIRTKEY, CONTROL, NOINVERT
    VK_BACK,        Undo_Item,              VIRTKEY, ALT, NOINVERT
    VK_DELETE,      Cut_Item,               VIRTKEY, SHIFT, NOINVERT
    VK_F1,          Help_Item,              VIRTKEY, NOINVERT
    VK_F2,          Cut_Item,               VIRTKEY, NOINVERT
    VK_F3,          Copy_Item,              VIRTKEY, NOINVERT
    VK_F4,          Paste_Item,             VIRTKEY, NOINVERT
    VK_HELP,        Help_Item,              VIRTKEY, CONTROL, NOINVERT
    VK_HELP,        Help_Item,              VIRTKEY, SHIFT, CONTROL, NOINVERT
    VK_INSERT,      Copy_Item,              VIRTKEY, CONTROL, NOINVERT
    VK_INSERT,      Paste_Item,             VIRTKEY, SHIFT, NOINVERT
    "W",            Close_Item,             VIRTKEY, CONTROL, NOINVERT
    "X",            Cut_Item,               VIRTKEY, CONTROL, NOINVERT
    "Z",            Undo_Item,              VIRTKEY, CONTROL, NOINVERT
END


////////////////////////////////////////////////////////////////////////////
//
// String Table
//

STRINGTABLE DISCARDABLE
BEGIN
    ProgName_Str            "WiniEdit"
    NoTitle_Str             "Untitled"
    FileFilter_Str          "WiniEdit files (*.wed)|*.wed|Plain text files
(*.txt)|*.txt|ASCII files (*.asc)|*.asc*|All text files (*.wed, *.txt,
*.asc)|*.wed;*.txt;*.asc|All files (*.*)|*.*||"
    FileExt_Str             "wed"
END

STRINGTABLE DISCARDABLE
BEGIN
    Save_Msg                "Save document ""%s"" before closing?"
    Revert_Msg              "Revert to most recently saved version of document
""%s""?"
    DefaultFormat_Msg       "Revert document ""%s"" to standard text format?"
    WrongType_Msg           "Sorry, WiniEdit works with text documents only. Can't
```

```
read or write document ""%s"".
    TooLong_Msg             "Sorry, document ""%s"" is too long for WiniEdit to read."
    OutOfMem_Msg            "Out of memory!"
    IOError_Msg             "Unanticipated input/output error #%s."
END


#ifndef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

Appendix B

MiniEdit Code Listing

Here, for comparison, is the complete code of my example Macintosh application program, MiniEdit, from which the Windows program WiniEdit was derived. The code was translated from the original Pascal version of the program as published in *Macintosh Revealed*, Volume Two. Again, the code uses C++-style comments and a few other syntactic conveniences, but is essentially written in straight C.

Make File

Here is **`MiniEdit.make`**, the MPW (Macintosh Programmer's Workshop) make file for the C version of MiniEdit:

```
#   File:       "MiniEdit.make"
#   Target:     "MiniEdit"
#   Sources:    "MiniEdit.cp" "MiniEdit.r"
#   Created:    Wednesday, October 26, 1994 3:58:40 PM


OBJECTS = "MiniEdit.cp.o"


"MiniEdit" ff "MiniEdit.make" "MiniEdit.r" "MiniEdit.rsrc"
   Rez "MiniEdit.r" -append -o "MiniEdit"

"MiniEdit" ff "MiniEdit.make" {OBJECTS}
   Link -t APPL -c MEDT ∂
      {OBJECTS} ∂
      #"{CLibraries}"CSANELib.o ∂
      #"{CLibraries}"Math.o ∂
      #"{CLibraries}"CplusLib.o ∂
      #"{CLibraries}"Complex.o ∂
      #"{CLibraries}"StdCLib.o ∂
      "{Libraries}"Runtime.o ∂
      "{Libraries}"Interface.o ∂
      -o "MiniEdit"
   SetFile "MiniEdit" -a B
"MiniEdit.cp.o" f "MiniEdit.make" "MiniEdit.cp"
    CPlus   "MiniEdit.cp"
```

Header File

Here is the program's header file, **`MiniEdit.h`**:

```
//
//                                  MiniEdit
//                        Example Macintosh application program
//                          S. Chernicoff    24 October 1994
//

//     Multiple-window text editor to illustrate event-driven program structure


//-----------------------------------------------------------------------------------------------------------


// Global Constants


   const short ChangeFlag = 0x0002;                  // Mask for extracting "change bit" from event modifiers

   const short MenuBarHeight = 20;                   // Height of menu bar in pixels
   const short TitleBarSize  = 18;                   // Size of window title bar in pixels
   const short ScreenMargin  =  4;                   // Width of "safety margin" around edge of screen

   const short MinWidth   = 80;                      // Minimum width of window in pixels
   const short MinHeight  = 80;                      // Minimum height of window in pixels
   const short SBarWidth  = 16;                      // Width of scroll bars in pixels
   const short TextMargin = 4;                       // Inset from window to text rectangle

   const short DlgTop  = 100;                        // Top edge of dialog box for Get and Put dialogs
   const short DlgLeft =  85;                        // Left edge of dialog box for Get and Put dialogs

   const short AppleID = 1;                          // Menu ID for Apple menu
      const short AboutItem  = 1;                       // Item number for About... command

   const short FileID  = 2;                          // Menu ID for File menu
      const short NewItem    =  1;                      // Item number for New command
      const short OpenItem   =  2;                      // Item number for Open... command
      const short CloseItem  =  3;                      // Item number for Close command
      const short SaveItem   =  5;                      // Item number for Save command
      const short SaveAsItem =  6;                      // Item number for Save As... command
      const short RevertItem =  7;                      // Item number for Revert to Saved command
      const short SetupItem  =  9;                      // Item number for Page Setup... command
      const short PrintItem  = 10;                      // Item number for Print... command
      const short QuitItem   = 12;                      // Item number for Quit command

   const short EditID  = 3;                          // Menu ID for Edit menu
      const short UndoItem   = 1;                       // Item number for Undo command
      const short CutItem    = 3;                       // Item number for Cut command
```

```
      const short CopyItem   = 4;                  // Item number for Copy command
      const short PasteItem  = 5;                  // Item number for Paste command
      const short ClearItem  = 7;                  // Item number for Clear command

   const short WindowID  = 1000;                  // Resource ID for window template
   const short ScrollID  = 1000;                  // Resource ID for scroll bar template
   const short NoTitleID = 1000;                  // Resource ID of title string for empty window

   const short AboutID     = 1000;                // Resource ID for About alert
   const short SaveID      = 1001;                // Resource ID for Save alert
   const short RevertID    = 1002;                // Resource ID for Revert alert
   const short CantPrintID = 1003;                // Resource ID for Can't Print alert
   const short WrongTypeID = 1004;                // Resource ID for Wrong Type alert
   const short TooLongID   = 1005;                // Resource ID for File Too Long alert
   const short OpWrID      = 1006;                // Resource ID for Already Open alert
   const short IOErrID     = 1007;                // Resource ID for I/O Error alert

//-------------------------------------------------------------------------------------------------------


// Global Types


   struct WindowData
      {
         TEHandle       editRec;                 // Handle to edit record
         ControlHandle  scrollBar;               // Handle to scroll bar
         Boolean        dirty;                   // Document changed since last saved?
         Byte           padding;                 // Extra byte for padding
         short          volNumber;               // Volume reference number
         short          fileNumber;              // File reference number

      }; /* end WindowData */


   typedef WindowData   *WDPtr,
                        **WDHandle;

//-------------------------------------------------------------------------------------------------------


// Global Variables


   EventRecord  TheEvent;                         // Current event

   WindowPtr      TheWindow    = nil;             // Pointer to currently active window
   ControlHandle  TheScrollBar = nil;             // Handle to active window's scroll bar
   TEHandle       TheText      = nil;             // Handle to active window's edit record

   Rect   ScreenBounds;                           // Boundary rectangle of screen
   short  ScreenWidth;                            // Width of screen in pixels
   short  ScreenHeight;                           // Height of screen in pixels
```

```
   short  OldMask;                              // Original value of system event mask

   MenuHandle  AppleMenu;                      // Handle to Apple menu
   MenuHandle  FileMenu;                       // Handle to File menu
   MenuHandle  EditMenu;                       // Handle to Edit menu

   CursHandle  IBeam;                          // Handle to I-beam cursor
   CursHandle  Watch;                          // Handle to wristwatch cursor

   short  OpenCount  = 0;                      // Number of windows opened
   short  CloseCount = 0;                      // Number of windows closed

   short    ScrapCompare;                      // Previous scrap count for comparison
   Boolean  ScrapDirty = false;                // Has scrap been changed?

   Boolean  Quitting  = false;                 // Closing up shop?
   Boolean  Finished  = false;                 // All closed?
   Boolean  ErrorFlag = false;                 // I/O error flag


//-------------------------------------------------------------------------------------------------------

// Function Prototypes

void Initialize (void);
      // Do one-time-only initialization.
   void SetUpMenus (void);
         // Set up menus.
   void SetUpCursors (void);
         // Set up cursors.
   void DoStartup (void);
         // Process Finder startup information.
void MainLoop (void);
      // Execute one pass of main program loop.
   void FixCursor (void);
         // Adjust cursor for region of screen.
   void DoEvent (void);
         // Get and process one event.
      void DoMouseDown (void);
            // Handle mouse-down event.
         void DoMenuClick (void);
               // Handle mouse-down event in menu bar.
            void DoMenuChoice (long menuChoice);
                  // Handle user's menu choice.
               void DoAppleChoice (short theItem);
                     // Handle choice from Apple menu.
                  void DoAbout (void);
                        // Handle About MiniEdit... command.
               void DoFileChoice (short theItem);
                     // Handle choice from File menu.
                  void DoNew (void);
                        // Handle New command.
                     void OffsetWindow (WindowPtr whichWindow);
                           // Offset location of new window.
```

```
        void DoOpen (void);
              // Handle Open... command.
           void OpenFile (Str255 fileName, short vNum);
                 // Open document file.
        void DoClose (void);
              // Handle Close command.
           void CloseAppWindow (void);
                 // Close application window.
           void CloseSysWindow (void);
                 // Close system window.
        void DoSave (void);
              // Handle Save command.
        void DoSaveAs (void);
              // Handle Save As... command.
           void WriteFile (short theFile, short volNum);
                 // Write window contents to a file.
        void DoRevert (void);
              // Handle Revert to Saved command.
        void DoSetup (void);
              // Handle Page Setup... command.
        void DoPrint (void);
              // Handle Print... command.
        void DoQuit (void);
              // Handle Quit command.
     void DoEditChoice (short theItem);
           // Handle choice from Edit menu.
        void DoUndo (void);
              // Handle Undo command.
        void DoCut (void);
              // Handle Cut command.
        void DoCopy (void);
              // Handle Copy command.
        void DoPaste (void);
              // Handle Paste command.
        void DoClear (void);
              // Handle Clear command.
void DoSysClick (WindowPtr whichWindow);
     // Handle mouse-down event in system window.
void DoContent (WindowPtr whichWindow);
     // Handle mouse-down event in content region of active window.
   void DoScroll (short thePart, Point thePoint);
         // Handle mouse-down event in scroll bar.
      pascal void ScrollText (ControlHandle theControl, short thePart);
            // Scroll text within window.
      void AdjustText (void);
            // Adjust text within window to match scroll bar setting.
      pascal Boolean AutoScroll (void);
            // Handle automatic scrolling during text selection.
   void DoSelect (Point thePoint);
         // Handle mouse-down event in text rectangle.
      void FixEditMenu (void);
            // Enable/disable editing commands.
void DoDrag (WindowPtr whichWindow);
     // Handle mouse-down event in drag region.
```

```
        void DoGrow (WindowPtr whichWindow);
              // Handle mouse-down event in size region.
           void FixScrollBar (void);
                 // Resize window's scroll bar.
           void FixText (void);
                 // Resize window's text rectangle.
        void DoGoAway (WindowPtr whichWindow);
              // Handle mouse-down event in close region.
        void DoZoom (WindowPtr whichWindow, short inOrOut);
              // Handle mouse-down event in zoom region.
     void DoKeystroke (void);
           // Handle keystroke.
     void DoTyping (unsigned char ch);
           // Handle character typed from keyboard.
     void DoUpdate (void);
           // Handle update event.
     void DoActivate (void);
           // Handle activate (or deactivate) event.
        void ActWindow (WindowPtr whichWindow);
              // Activate window.
        void DeactWindow (WindowPtr whichWindow);
              // Deactivate window.
void Finalize (void);
     // Do one-time-only finalization.
   void WindowDirty (Boolean isDirty);
         // Mark window dirty or clean.
   void AdjustScrollBar (void);
         // Adjust scroll bar to length of document.
   void ScrollToSelection (void);
         // Scroll current selection into view.
   void ScrollCharacter (short theCharacter, Boolean toBottom);
         // Scroll character into view.
   void ReadDeskScrap (void);
         // Copy desk scrap to Toolbox scrap.
   void WriteDeskScrap (void);
         // Copy Toolbox scrap to desk scrap.
   void IOCheck (OSErr resultCode);
         // Check for I/O error.
```

Code File

This is the main code file, **`MiniEdit.cp`**, containing the C++ source code for the MiniEdit program:

```
//
//                                MiniEdit
//                    Example Macintosh application program
//                      S. Chernicoff    24 October 1994
//

//     Multiple-window text editor to illustrate event-driven program structure
```

```
   #include <SysEqu.h>                          // System globals
   #include <Types.h>                           // Elementary data types
   #include <QuickDraw.h>                       // QuickDraw graphics routines
   #include <Events.h>                          // Event-handling routines
   #include <Windows.h>                         // Window-handling routines
   #include <Controls.h>                        // Control-handling routines
   #include <Menus.h>                           // Menu-handling routines
   #include <Dialogs.h>                         // Dialog-handling routines
   #include <TextEdit.h>                        // Text-editing routines
   #include <Fonts.h>                           // Font-handling routines
   #include <Desk.h>                            // Desk accessory support routines
   #include <Scrap.h>                           // Clipboard-management routines
   #include <Files.h>                           // File-handling routines
   #include <StandardFile.h>                    // Standard File Package
   #include <Memory.h>                          // Memory-management routines
   #include <SegLoad.h>                         // Segment Loader routines
   #include <ToolUtils.h>                       // Toolbox utility routines
   #include <OSUtils.h>                         // Operating System utility routines
   #include <Errors.h>                          // Error codes

   #include "MiniEdit.h"


//-------------------------------------------------------------------------------------------------------

main ()

   // Main program

   {
      Initialize ();                            // Do one-time-only initialization

      do
         MainLoop ();                           // Execute one pass of main loop
      while ( !Finished );                      // Continue until time to quit

      Finalize ();                              // Do one-time-only finalization

      return noErr;                             // Signal successful completion

   }  /* end main */

//-------------------------------------------------------------------------------------------------------

void Initialize (void)

   // Do one-time-only initialization.

   {
      short        masterBlocks = 4;            // Number of master pointer blocks to preallocate
      short       *sysMaskPtr;                  // Pointer for finding old event mask
      short        newMask;                     // New value for system event mask
```

```
    PScrapStuff  scrapInfo;                        // Pointer to scrap information record


    InitGraf ( Ptr(&qd.thePort) );                 // Initialize QuickDraw
    InitFonts ();                                  // Initialize fonts
    InitWindows ();                                // Initialize windows
    InitMenus ();                                  // Initialize menus
    TEInit ();                                     // Initialize text editing
    InitDialogs (nil);                             // Initialize dialogs

    MaxApplZone ();                                // Expand heap to maximum size
    while ( (masterBlocks--) > 0 )                 // Preallocate master pointers to
       MoreMasters ();                             //    minimize heap fragmentation

    ScreenBounds = qd.screenBits.bounds;                    // Get boundary rectangle of screen
    ScreenWidth  = ScreenBounds.right  - ScreenBounds.left;  // Set screen dimensions
    ScreenHeight = ScreenBounds.bottom - ScreenBounds.top;

    sysMaskPtr = (short*)(SysEvtMask);             // Point to system event mask
    OldMask    = *sysMaskPtr;                      // Save old mask value

    newMask = everyEvent - keyUpMask - mUpMask;    // Disable key-up and mouse-up events
    SetEventMask (newMask);                        // Set the mask
    FlushEvents  (everyEvent, 0);                  // Clear out event queue

    SetUpMenus ();                                 // Create program's menus
    SetUpCursors ();                               // Get standard cursors

    scrapInfo    = InfoScrap();                    // Get scrap info
    ScrapCompare = scrapInfo->scrapCount + 1;      // Force scrap transfer
    ReadDeskScrap ();                              // Read desk scrap into Toolbox scrap

    DoStartup ();                                  // Process Finder startup information

  }  /* end Initialize */

//--------------------------------------------------------------------------------------------------

void SetUpMenus (void)

  // Set up menus.

  {
    AppleMenu = GetMenu(AppleID);                  // Get Apple menu from resource file
    AddResMenu (AppleMenu, 'DRVR');                // Add names of available desk accessories
    InsertMenu (AppleMenu, 0);                     // Install at end of menu bar

    FileMenu = GetMenu(FileID);                    // Get File menu from resource file
    InsertMenu (FileMenu, 0);                      // Install at end of menu bar

    EditMenu = GetMenu(EditID);                    // Get Edit menu from resource file
    InsertMenu (EditMenu, 0);                      // Install at end of menu bar

    DrawMenuBar ();                                // Show new menu bar on screen
```

```
   }  /* end SetUpMenus */

//-------------------------------------------------------------------------------------------------------

void SetUpCursors (void)

   // Set up cursors.

   {
      IBeam = GetCursor(iBeamCursor);                 // Get cursors from system resource file
      Watch = GetCursor(watchCursor);

      InitCursor ();                                  // Set standard arrow cursor

   }  /* end SetUpCursors */

//-------------------------------------------------------------------------------------------------------

void DoStartup (void)

   // Process Finder startup information.

   {
      short      theMessage;                          // Open or print?
      short      nDocs;                               // Number of documents selected in Finder
      short      thisDoc;                             // Index number of document
      AppFile    docInfo;                             // Startup information about one document
      StringPtr  fileName;                            // Name of document to be opened
      short      volNum;                              // Reference number of document's volume
      OSType     fileType;                            // Document's file type


      CountAppFiles (&theMessage, &nDocs);            // Get number of documents and startup message

      if (theMessage == appPrint)                     // Did user choose Print in Finder?
         {
            StopAlert (CantPrintID, nil);             // Post alert
            ExitToShell ();                           // Return to Finder

         }  /* end if (theMessage == appPrint) */

      if ( nDocs == 0 )                               // If no documents selected,
         DoNew ();                                    //    just open an empty window

      else
         for ( (thisDoc = 1); (thisDoc <= nDocs); (thisDoc++) )   // Otherwise loop through documents
            {
               GetAppFiles (thisDoc, &docInfo);       // Get startup information

               fileName = docInfo.fName;              // Get file name
               volNum   = docInfo.vRefNum;            // Get volume reference number
               fileType = docInfo.fType;              // Get file type
```

```
            if ( fileType == 'TEXT' )              // Is it a text file?
               {
                  ErrorFlag = false;               // Clear I/O error flag

                  OpenFile (fileName, volNum);     // Read file into a window
                  ClrAppFiles (thisDoc);           // Tell Finder it's been processed

               }  /* end if ( fileType == 'TEXT' ) */

            else
               {
                  ParamText (fileName, "\p", "\p", "\p");   // Merge in file name
                  InitCursor ();                   // Set arrow cursor
                  StopAlert (WrongTypeID, nil);    // Post alert

               }  /* end else */

         }  /* end for (thisDoc) */

   }  /* end DoStartup */

//--------------------------------------------------------------------------------------------------------

void MainLoop (void)

   // Execute one pass of main program loop.

   {
      if ( FrontWindow() == nil )                  // Is the desktop empty?
         {
            DisableItem (EditMenu, UndoItem);      // Disable inapplicable menu commands
            DisableItem (EditMenu, CutItem);
            DisableItem (EditMenu, CopyItem);
            DisableItem (EditMenu, PasteItem);
            DisableItem (EditMenu, ClearItem);

            DisableItem (FileMenu, CloseItem);
            DisableItem (FileMenu, SaveItem);
            DisableItem (FileMenu, SaveAsItem);
            DisableItem (FileMenu, RevertItem);
            DisableItem (FileMenu, SetupItem);
            DisableItem (FileMenu, PrintItem);

         }  /* end if ( FrontWindow() == nil ) */

      FixCursor ();                                // Adjust cursor for region of screen
      SystemTask ();                               // Do system idle processing

      if ( TheText != nil )
         TEIdle (TheText);                         // Blink cursor

      DoEvent ();                                  // Get and process one event

   }  /* end MainLoop */
```

```
//--------------------------------------------------------------------------------------------------

void FixCursor (void)

   // Adjust cursor for region of screen.

   {
      Point  mousePoint;                        // Current mouse position in window coordinates
      Rect   textRect;                          // Active window's text rectangle


      if ( Quitting )                           // Skip cursor adjustment during quit sequence
         return;

      if ( FrontWindow() == nil )               // Screen empty?
         InitCursor ();                         // Set arrow cursor

      else if ( FrontWindow() == TheWindow )    // Is one of our windows active?
         {
            GetMouse (&mousePoint);             // Get mouse position
            textRect = (**TheText).viewRect;    // Get window's text rectangle

            if ( PtInRect(mousePoint, &textRect) )    // Is mouse in text rectangle?
               SetCursor ( &(**IBeam) );        // Set I-beam cursor
            else
               InitCursor ();                   // Set arrow cursor

         }  /* end if ( FrontWindow() == TheWindow ) */

      else                                      // A system window is active:
         /* Do nothing */;                      //    let desk accessory set cursor

   }  /* end FixCursor */

//--------------------------------------------------------------------------------------------------

void DoEvent (void)

   // Get and process one event.

   {
      ErrorFlag = false;                        // Clear I/O error flag

      if ( GetNextEvent(everyEvent, &TheEvent) )      // Get next event

         switch ( TheEvent.what )
            {
               case mouseDown:
                  if ( !Quitting )
                     DoMouseDown();             // Handle mouse-down event
                  break;

               case keyDown:
```

```
            case autoKey:
               if ( !Quitting )
                  DoKeystroke ();                  // Handle keystroke
               break;

            case updateEvt:
               DoUpdate ();                        // Handle update event
               break;

            case activateEvt:
               DoActivate ();                      // Handle activate/deactivate event
               break;

            default:
               break;

         }  /* end switch ( TheEvent.what ) */

      else if ( Quitting && (TheEvent.what == nullEvent) )  // Closing up shop after a Quit command?
         {
            if ( FrontWindow() != nil )            // Any windows on the screen?
               DoClose ();                         // Close the frontmost
            else
               Finished = true;                    // Signal end of program

         }  /* end if ( Quitting && (TheEvent.what == nullEvent) ) */

   }  /* end DoEvent */

//--------------------------------------------------------------------------------------------------------

void DoMouseDown (void)

   // Handle mouse-down event.

   {
      WindowPtr  whichWindow;                      // Window where mouse was pressed
      short      thePart;                          // Part of screen where mouse was pressed


      thePart = FindWindow(TheEvent.where, &whichWindow);   // Where on the screen was mouse pressed?

      switch ( thePart )
         {
            case inDesk:
               break;                              // Nothing to do for click on desktop

            case inMenuBar:
               DoMenuClick ();                     // Handle click in menu bar
               break;

            case inSysWindow:
               DoSysClick (whichWindow);           // Handle click in system window
               break;
```

```
            case inContent:
               DoContent (whichWindow);              // Handle click in content region
               break;

            case inDrag:
               DoDrag (whichWindow);                 // Handle click in drag region
               break;

            case inGrow:
               DoGrow (whichWindow);                 // Handle click in size region
               break;

            case inGoAway:
               DoGoAway (whichWindow);               // Handle click in close region
               break;

            case inZoomIn:
               DoZoom (whichWindow, inZoomIn);       // Handle click in zoom region
               break;

            case inZoomOut:
               DoZoom (whichWindow, inZoomOut);      // Handle click in zoom region
               break;

         }  /* end switch ( thePart ) */

   }  /* end DoMouseDown */

//--------------------------------------------------------------------------------------------------------

void DoMenuClick (void)

   // Handle mouse-down event in menu bar.

   {
      long  menuChoice;                              // Menu ID and item number


      menuChoice = MenuSelect(TheEvent.where);       // Track mouse
      DoMenuChoice (menuChoice);                     // Handle user's menu choice


   }  /* end DoMenuClick */

//--------------------------------------------------------------------------------------------------------

void DoMenuChoice (long menuChoice)

   // Handle user's menu choice.

   {
      const short  noMenu = 0;                       // No menu selected
```

```
      short  theMenu = HiWord(menuChoice);            // Menu ID of selected menu
      short  theItem = LoWord(menuChoice);            // Item number of selected item


      switch ( theMenu )
         {
            case noMenu:
               break;                                 // No menu selected, nothing to do

            case AppleID:
               DoAppleChoice (theItem);               // Handle choice from Apple menu
               break;

            case FileID:
               DoFileChoice (theItem);                // Handle choice from File menu
               break;

            case EditID:
               DoEditChoice (theItem);                // Handle choice from Edit menu
               break;

         }  /* end switch ( theMenu ) */

      HiliteMenu (0);                                 // Unhighlight menu title

   }  /* end DoMenuChoice */

//--------------------------------------------------------------------------------------------------------------

void DoAppleChoice (short theItem)

   // Handle choice from Apple menu.

   {
      Str255  accName;                                // Name of desk accessory


      switch ( theItem )
         {
            case AboutItem:
               DoAbout ();                            // Handle About MiniEdit... command
               break;

            default:
               {
                  if ( FrontWindow() == nil )         // Is the desktop empty?
                     {
                        EnableItem (FileMenu, CloseItem);   // Enable Close command

                        EnableItem (EditMenu, UndoItem);    // Enable standard
                        EnableItem (EditMenu, CutItem);     //    editing commands
                        EnableItem (EditMenu, CopyItem);    //    for desk accessory
                        EnableItem (EditMenu, PasteItem);
                        EnableItem (EditMenu, ClearItem);
```

```
                  }  /* end if ( FrontWindow() == nil ) */

               GetItem (AppleMenu, theItem, accName);    // Get accessory name
               OpenDeskAcc (accName);                    // Open desk accessory

               break;

            }  /* end default */

         }  /* end switch ( theItem ) */

   }  /* end DoAppleChoice */

//--------------------------------------------------------------------------------------------------------------

void DoAbout (void)

   // Handle About MiniEdit... command.

   {
      InitCursor ();                                 // Set arrow cursor
      Alert (AboutID, nil);                          // Post alert

   }  /* end DoAbout */

//--------------------------------------------------------------------------------------------------------------

void DoFileChoice (short theItem)

   // Handle choice from File menu.

   {
      switch ( theItem )
         {
            case NewItem:
               DoNew ();                             // Handle New command
               break;

            case OpenItem:
               DoOpen ();                            // Handle Open... command
               break;

            case CloseItem:
               DoClose ();                           // Handle Close command
               break;

            case SaveItem:
               DoSave ();                            // Handle Save command
               break;

            case SaveAsItem:
               DoSaveAs ();                          // Handle Save As... command
               break;
```

```
            case RevertItem:
               DoRevert ();                          // Handle Revert to Saved command
               break;

            case SetupItem:
               DoSetup ();                           // Handle Page Setup... command
               break;

            case PrintItem:
               DoPrint ();                           // Handle Print... command
               break;

            case QuitItem:
               DoQuit ();                            // Handle Quit command
               break;

         }  /* end switch ( theItem ) */

   }  /* end DoFileChoice */

//--------------------------------------------------------------------------------------------------

void DoNew (void)

   // Handle New command.

   {
      WDHandle  theData;                             // Handle to window's data record
      Handle    dataHandle;                          // Untyped handle for creating data record
      Rect      destRect;                            // Wrapping rectangle for window's text
      Rect      viewRect;                            // Clipping rectangle for window's text


      TheWindow = GetNewWindow(WindowID, nil, WindowPtr(-1));  // Make new window from template

      OffsetWindow (TheWindow);                      // Offset from location of previous window
      ShowWindow   (TheWindow);                      // Make window visible

      SetPort  (TheWindow);                          // Get into the window's port
      TextFont (geneva);                             // Set text font
      TextSize (9);                                  // Set text size

      SetRect (&viewRect, 0,                         // Set up clipping rectangle
                          0,
                          (*TheWindow).portRect.right  - (SBarWidth - 1),
                          (*TheWindow).portRect.bottom - (SBarWidth - 1));
      destRect = viewRect;                           // Set up wrapping rectangle
      InsetRect (&destRect, TextMargin, 0);          // Inset by text margin

      dataHandle = NewHandle( sizeof(WindowData) );  // Allocate window data record
      theData    = WDHandle(dataHandle);             // Convert to typed handle
      SetWRefCon (TheWindow, long(theData));         // Store as reference constant
```

```
      TheText = TENew (&destRect, &viewRect);         // Make edit record
      (**theData).editRec = TheText;                  // Store handle in data record
      SetClikLoop (AutoScroll, TheText);              // Install auto-scroll routine

      TheScrollBar = GetNewControl(ScrollID, TheWindow); // Make scroll bar
      (**theData).scrollBar = TheScrollBar;           // Store handle in data record

      (**theData).dirty      = false;                 // Document is initially clean
      (**theData).fileNumber = 0;                     // Window has no associated file
      (**theData).volNumber  = 0;                     //    or volume

      EnableItem (FileMenu, CloseItem);               // Enable Close command on menu

   }  /* end DoNew */

//--------------------------------------------------------------------------------------------------------

void OffsetWindow (WindowPtr whichWindow)

   // Offset location of new window.

   {
      const short  hOffset = 20;                      // Horizontal offset from previous window, in pixels
      const short  vOffset = 20;                      // Vertical offset from previous window, in pixels

      short  windowWidth;                             // Width of window in pixels
      short  windowHeight;                            // Height of window in pixels
      short  hExtra;                                  // Excess screen width in pixels
      short  vExtra;                                  // Excess screen height in pixels
      short  hMax;                                    // Maximum number of windows horizontally
      short  vMax;                                    // Maximum number of windows vertically
      short  windowLeft;                              // Left edge of window in global coordinates
      short  windowTop;                               // Top edge of window in global coordinates


      windowWidth   = whichWindow->portRect.right  - whichWindow->portRect.left; // Get window dimensions
      windowHeight  = whichWindow->portRect.bottom - whichWindow->portRect.top;  //    from port rectangle
      windowHeight += TitleBarSize;                                              // Adjust for title bar

      hExtra =  ScreenWidth  - windowWidth;                    // Find excess screen width
      vExtra = (ScreenHeight - MenuBarHeight) - windowHeight;  // Find excess screen height

      hMax = (hExtra / hOffset) + 1;                           // Find maximum number of windows horizontally
      vMax = (vExtra / vOffset) + 1;                           // Find maximum number of windows vertically

      OpenCount++;                                             // Increment open window count
      windowLeft = (OpenCount % hMax) * hOffset;               // Calculate horizontal
      windowTop  = (OpenCount % vMax) * vOffset;               //    and vertical offset

      windowLeft += TitleBarSize;                              // Adjust for title bar
      windowTop  += MenuBarHeight;                             //    and menu bar

      MoveWindow (whichWindow, windowLeft, windowTop, false);  // Move window to new location
```

```
   }  /* end OffsetWindow */

//--------------------------------------------------------------------------------------------------------------

void DoOpen (void)

   // Handle Open... command.

   {
      Point       dlgOrigin;                        // Top-left corner of dialog box
      SFTypeList  theTypeList;                      // List of file types to display
      SFReply     theReply;                         // Data returned by Get dialog


      SetPt (&dlgOrigin, DlgLeft, DlgTop);          // Set up dialog origin
      theTypeList[0] = 'TEXT';                      // Display text files only

      SFGetFile (dlgOrigin, "\p", nil, 1, theTypeList, nil, &theReply); // Get file name from user

      if ( theReply.good )                          // Did user confirm file selection?
         OpenFile (theReply.fName, theReply.vRefNum); // Open file and read into window

   }  /* end DoOpen */

//--------------------------------------------------------------------------------------------------------------

void OpenFile (Str255 fileName, short vNum)

   // Open document file.

   {
      long      theRefCon;                          // Window's reference constant
      WDHandle  theData;                            // Handle to window data record
      short     theFile;                            // Reference number of file
      OSErr     resultCode;                         // I/O error code


      resultCode = FSOpen (fileName, vNum, &theFile); // Open the file
      IOCheck (resultCode);                         // Check for error
      if (ErrorFlag) return;                        // On error, exit to main event loop

      DoNew ();                                     // Open a new window

      theRefCon = GetWRefCon(TheWindow);            // Get reference constant
      theData   = WDHandle(theRefCon);              // Convert to data handle

      (**theData).volNumber  = vNum;                // Save volume and file number
      (**theData).fileNumber = theFile;             //    in window data record
      SetWTitle (TheWindow, fileName);              // File name becomes window title

      DoRevert ();                                  // Read file into window

      if ( ErrorFlag )                              // Error reading file?
         CloseAppWindow ();                         // Close and dispose of the window
```

```
   }  /* end OpenFile */

//-----------------------------------------------------------------------------------------------------------

void DoClose (void)

   // Handle Close command.

   {
      if ( FrontWindow() == TheWindow )              // Is the active window one of ours?
         CloseAppWindow ();                          // Close application window
      else
         CloseSysWindow ();                          // Close system window

   }  /* end DoClose */

//-----------------------------------------------------------------------------------------------------------

void CloseAppWindow (void)

   // Close application window.

   {
      const short saveItem    = 1;                   // Item number for Save button
      const short discardItem = 2;                   // Item number for Discard button
      const short cancelItem  = 3;                   // Item number for Cancel button

      WindowPtr  thisWindow;                         // Pointer to window being closed
      long       theRefCon;                          // Window's reference constant
      WDHandle   theData;                            // Handle to window data record
      Handle     dataHandle;                         // Untyped handle for destroying data record
      Str255     theTitle;                           // Title of window
      short      theFile;                            // Reference number of window's file
      TEHandle   theEditRec;                         // Handle to window's edit record
      short      theItem;                            // Item number for Save alert
      OSErr      resultCode;                         // I/O error code


      theRefCon = GetWRefCon(TheWindow);             // Get reference constant
      theData   = WDHandle(theRefCon);               // Convert to data handle

      if ( (**theData).dirty )                       // Have window contents been changed?
         {
            GetWTitle (TheWindow, theTitle);         // Get window title
            ParamText (theTitle, "\p", "\p", "\p");  // Substitute into alert text

            InitCursor ();                           // Set arrow cursor
            theItem = CautionAlert(SaveID, nil);     // Post alert
            switch ( theItem )
               {
                  case saveItem:
                     DoSave ();                      // Save window contents to disk
                     if ( ErrorFlag )                // Check for I/O error
```

```
                  return;                    // Exit to main event loop
               break;

            case discardItem:
               break;                        // Nothing to do

            case cancelItem:
               Quitting = false;             // Cancel Quit command, if any
               return;                       // Exit to main event loop

         }  /* end switch ( theItem ) */

      }  /* end if ( (**theData).dirty ) */

   theFile = (**theData).fileNumber;          // Get file reference number
   if ( theFile != 0 )                        // Is window associated with a file?
      {
         resultCode = FSClose(theFile);       // Close file
         IOCheck (resultCode);                // Post error alert, if any

      }  /* end if ( theFile != 0 ) */

   thisWindow = TheWindow;                    // Save window pointer (DoActivate will change TheWindow)
   HideWindow (TheWindow);                    // Force deactivate event

   if ( GetNextEvent(activMask, &TheEvent) )  // Get deactivate event
      DoActivate ();                          //    and handle it
   if ( GetNextEvent(activMask, &TheEvent) )  // Get activate event
      DoActivate ();                          //    and handle it

   CloseCount++;                              // Increment closed window count
   if ( CloseCount == OpenCount )             // Closing last application window on screen?
      {
         OpenCount  = 0;                      // Reset window offset to zero
         CloseCount = 0;

      }  /* end if ( CloseCount == OpenCount ) */

   theEditRec = (**theData).editRec;          // Get handle to edit record
   TEDispose (theEditRec);                    // Dispose of edit record

   dataHandle = Handle(theData);              // Convert to raw handle
   DisposHandle (dataHandle);                 // Dispose of window data record

   DisposeWindow (thisWindow);                // Dispose of window and scroll bar

}  /* end CloseAppWindow */

//--------------------------------------------------------------------------------------------------------

void CloseSysWindow (void)

   // Close sys window.
```

```
   {
      WindowPeek  whichWindow;                  // Pointer for access to window's fields
      short       accNumber;                    // Reference number of desk accessory


      whichWindow = WindowPeek( FrontWindow() );   // Convert to a WindowPeek

      accNumber = whichWindow->windowKind;      // Get reference number of desk accessory
      CloseDeskAcc (accNumber);                 // Close desk accessory

   }  /* end CloseSysWindow */

//--------------------------------------------------------------------------------------------------------

void DoSave (void)

   // Handle Save command.

   {
      long      theRefCon;                      // Window's reference constant
      WDHandle  theData;                        // Handle to window data record
      short     theVolume;                      // Volume reference number
      short     theFile;                        // File reference number


      theRefCon = GetWRefCon(TheWindow);        // Get reference constant
      theData   = WDHandle(theRefCon);          // Convert to data handle

      theVolume = (**theData).volNumber;        // Get volume reference number
      theFile   = (**theData).fileNumber;       // Get file reference number

      if ( theFile == 0 )                       // Is window associated with a file?
         DoSaveAs ();                           // Get file name from user
      else
         WriteFile (theFile, theVolume);        // Write to window's file

   }  /* end DoSave */

//--------------------------------------------------------------------------------------------------------

void DoSaveAs (void)

   // Handle Save As... command.

   {
      Point         dlgOrigin;                  // Top-left corner of dialog box
      SFReply       theReply;                   // Data returned by Put dialog

      short         oldFile;                    // Reference number of window's previous file
      short         newFile;                    // Reference number of file being saved
      short         newVol;                     // Volume reference number for file being saved
      StringPtr     newName;                    // Name of file being saved
      FInfo         theInfo;                    // File's Finder information
```

```
long          theRefCon;                        // Window's reference constant
WDHandle      theData;                          // Handle to window data record
Str255        theTitle;                         // Title of window

StringHandle  untitled;                         // Handle to title string for empty window
OSErr         resultCode;                       // I/O error code


SetPt     (&dlgOrigin, DlgLeft, DlgTop);        // Set up dialog origin
GetWTitle (TheWindow, theTitle);                // Get current file name from window title
SFPutFile (dlgOrigin, "\pSave under what name?", theTitle, nil, &theReply);
                                                // Get new file name from user

if ( !(theReply.good) )                         // Did user confirm file selection?
   {
      Quitting  = false;                        // Cancel Quit command, if any
      ErrorFlag = true;                         // Force exit to main event loop
      return;                                   // Skip rest of operation
   }  /* end if ( !(theReply.good) ) */

newName = theReply.fName;                       // Get file name
newVol  = theReply.vRefNum;                     // Get volume reference number

resultCode = GetFInfo (newName, newVol, &theInfo); // Get Finder info
switch ( resultCode )
   {
      case noErr:                               // File already exists
         if ( theInfo.fdType != 'TEXT' )        // Not a text file?
            {
               ParamText (newName, "\p", "\p", "\p"); // Substitute file name into text of alert
               StopAlert (WrongTypeID, nil);    // Post alert

               ErrorFlag = true;                // Force exit to main event loop
               return;                          // Skip rest of operation

            }  /* end if ( theInfo.fdType != 'TEXT' ) */
         break;

      case fnfErr:                              // File not found
         resultCode = Create (newName, newVol, 'MEDT', 'TEXT');   // Create the file
         IOCheck (resultCode);                  // Check for error
         if (ErrorFlag) return;                 // On error, exit to main event loop
         break;

      default:                                  // Unanticipated error
         IOCheck (resultCode);                  // Post error alert
         return;                                // Exit to main event loop

   }  /* end switch ( resultCode ) */

theRefCon = GetWRefCon(TheWindow);              // Get reference constant
theData   = WDHandle(theRefCon);                // Convert to data handle

SetCursor ( &(**Watch) );                       // Indicate delay
```

```
      oldFile = (**theData).fileNumber;              // Get file reference number from data record
      if ( oldFile != 0 )                            // Does window already have a file?
         {
            resultCode = FSClose (oldFile);          // Close old file
            IOCheck (resultCode);                    // Check for error
            if (ErrorFlag) return;                   // On error, exit to main event loop

         }  /* end if ( oldFile != 0 ) */

      resultCode = FSOpen (newName, newVol, &newFile);   // Open new file
      IOCheck (resultCode);                          // Check for error
      if (ErrorFlag)                                 // Error detected during open?
         {
            (**theData).volNumber  = 0;              // Window is left with no file: clear volume
            (**theData).fileNumber = 0;              //    and file numbers in window data

            untitled = GetString (NoTitleID);        // Get string from resource file
            newName  = &(**untitled);                // Convert from handle

         }  /* end if (ErrorFlag) */

      else
         {
            (**theData).volNumber  = newVol;         // Save new volume and file
            (**theData).fileNumber = newFile;        //    numbers in window data

            WriteFile (newFile, newVol);             // Write window's contents to file

         }  /* end else */

      SetWTitle (TheWindow, newName);                // Set new window title

   }  /* end DoSaveAs */

//-------------------------------------------------------------------------------------------------------

void WriteFile (short theFile, short volNum)

   // Write window contents to a file.

   {
      Handle  textHandle;                            // Handle to text of file
      long    textLength;                            // Length of text in bytes
      OSErr   resultCode;                            // I/O error code


      SetCursor ( &(**Watch) );                      // Indicate delay

      textHandle = (**TheText).hText;                // Get text handle and current length
      textLength = (**TheText).teLength;             //    from edit record

      resultCode = SetFPos (theFile, fsFromStart, 0); // Reset mark to beginning of file
      IOCheck (resultCode);                          // Check for error
```

```
      if (ErrorFlag) return;                          // On error, exit to main event loop

      MoveHHi (textHandle);                           // Move text to top of heap
      HLock   (textHandle);                           // Lock text
         resultCode = FSWrite ( theFile, &textLength, &(**textHandle) );   // Write text to file
      HUnlock (textHandle);                           // Unlock text
      IOCheck (resultCode);                           // Check for error
      if (ErrorFlag) return;                          // On error, exit to main event loop

      resultCode = SetEOF (theFile, textLength);      // Set length of file
      IOCheck (resultCode);                           // Check for error
      if (ErrorFlag) return;                          // On error, exit to main event loop

      resultCode = FlushVol (nil, volNum);            // Flush volume buffer
      IOCheck (resultCode);                           // Check for error
      if (ErrorFlag) return;                          // On error, exit to main event loop

      WindowDirty (false);                            // Mark window as clean

   }  /* end WriteFile */

//--------------------------------------------------------------------------------------------------------

void DoRevert (void)

   // Handle Revert to Saved command.

   {
      const short  maxLength = 32767;                 // Maximum document length in bytes

      long      theRefCon;                            // Window's reference constant
      WDHandle  theData;                              // Handle to window data record
      Str255    fileName;                             // Title of window
      short     theFile;                              // Reference number of file
      Handle    textHandle;                           // Text handle from window's edit record
      long      textLength;                           // Length of file in bytes
      short     theItem;                              // Item number returned by alert
      OSErr     resultCode;                           // I/O error code


      theRefCon = GetWRefCon(TheWindow);              // Get reference constant
      theData   = WDHandle(theRefCon);                // Convert to data handle

      if ( (**theData).dirty )                        // Have window contents been changed?
         {
            GetWTitle (TheWindow, fileName);          // Get file name from window title
            ParamText (fileName, "\p", "\p", "\p");   // Substitute into text of alert

            InitCursor ();                            // Set arrow cursor
            theItem = CautionAlert (RevertID, nil);   // Post alert

            if ( theItem == cancel )                  // Did user cancel?
               {
                  ErrorFlag = true;                   // Force exit to main event loop
```

```
            return;                          // Skip rest of operation

         }  /* end if ( theItem == cancel ) */

      }  /* end if ( (**theData).dirty ) */

   SetCursor ( &(**Watch) );                 // Indicate delay

   theFile = (**theData).fileNumber;         // Get file reference number from data record
   resultCode = GetEOF ( theFile, &textLength );   // Get length of file
   if ( textLength > maxLength )             // File too long?
      {
         GetWTitle (TheWindow, fileName);    // Get file name from window title
         ParamText (fileName, "\p", "\p", "\p");   // Substitute into text of alert

         InitCursor ();                      // Set arrow cursor
         theItem = StopAlert (TooLongID, nil);     // Post alert

         ErrorFlag = true;                   // Force exit

      }  /* end if ( textLength > maxLength ) */
   else
      IOCheck (resultCode);                  // Check for I/O error
   if (ErrorFlag) return;                    // On error, exit to main event loop

   resultCode = SetFPos (theFile, fsFromStart, 0); // Set mark at beginning of file
   IOCheck (resultCode);                     // Check for error
   if (ErrorFlag) return;                    // On error, exit to main event loop

   textHandle = (**TheText).hText;           // Get text handle from edit record
   SetHandleSize (textHandle, textLength);   // Adjust text to length of file
   (**TheText).teLength = textLength;        // Set text length in edit record

   MoveHHi (textHandle);                     // Move block to top of heap
   HLock   (textHandle);                     // Lock text handle
      resultCode = FSRead ( theFile, &textLength, &(**textHandle) ); // Read text of file into block
   HUnlock (textHandle);                     // Unlock text handle
   IOCheck (resultCode);                     // Check for error
   if (ErrorFlag) return;                    // On error, exit to main event loop

   TECalText (TheText);                      // Wrap text to window
   AdjustScrollBar ();                       // Adjust scroll bar to length of text
   TESetSelect (0, 0, TheText);              // Set insertion point at beginning

   InvalRect ( &(TheWindow->portRect) );     // Force update to redraw text
   WindowDirty (false);                      // Mark window as clean

}  /* end DoRevert */

//--------------------------------------------------------------------------------------------------

void DoSetup (void)

   // Handle Page Setup... command.
```

```
   {
      SysBeep (1);                                  // Page Setup... command not implemented

   }  /* end DoSetup */

//--------------------------------------------------------------------------------------------------------

void DoPrint (void)

   // Handle Print... command.

   {
      SysBeep (1);                                  // Print... command not implemented

   }  /* end DoPrint */

//--------------------------------------------------------------------------------------------------------

void DoQuit (void)

   // Handle Quit command.

   {
      Quitting = true;                              // Start closing down windows

   }  /* end DoQuit */

//--------------------------------------------------------------------------------------------------------

void DoEditChoice (short theItem)

   // Handle choice from Edit menu.

   {
      const short  undoCmd  = 0;                    // Code representing Undo command
      const short  cutCmd   = 2;                    // Code representing Cut command
      const short  copyCmd  = 3;                    // Code representing Copy command
      const short  pasteCmd = 4;                    // Code representing Paste command
      const short  clearCmd = 5;                    // Code representing Clear command


      switch ( theItem )
         {
            case UndoItem:
               if ( !SystemEdit(undoCmd) )          // Intercepted by a desk accessory?
                  DoUndo ();                        // Handle Undo command
               break;

            case CutItem:
               if ( !SystemEdit(cutCmd) )           // Intercepted by a desk accessory?
                  DoCut ();                         // Handle Cut command
               break;
```

```
            case CopyItem:
               if ( !SystemEdit(copyCmd) )          // Intercepted by a desk accessory?
                  DoCopy ();                        // Handle Copy command
               break;

            case PasteItem:
               if ( !SystemEdit(pasteCmd) )         // Intercepted by a desk accessory?
                  DoPaste ();                       // Handle Paste command
               break;

            case ClearItem:
               if ( !SystemEdit(clearCmd) )         // Intercepted by a desk accessory?
                  DoClear ();                       // Handle Clear command
               break;

         }  /* end switch ( theItem ) */

   }  /* end DoEditChoice */

//--------------------------------------------------------------------------------------------------------------

void DoUndo (void)

   // Handle Undo command.

   {
      SysBeep (1);                                  // Undo command not implemented

   }  /* end DoUndo */

//--------------------------------------------------------------------------------------------------------------

void DoCut (void)

   // Handle Cut command.

   {
      ScrollToSelection ();                         // Make sure selection is visible

      TECut (TheText);                              // Cut the selection

      AdjustScrollBar ();                           // Adjust scroll bar to length of text
      AdjustText ();                                // Adjust text to match scroll bar
      ScrollToSelection ();                         // Keep insertion point visible

      DisableItem (EditMenu, CutItem);              // Disable menu items that operate
      DisableItem (EditMenu, CopyItem);             //    on a nonempty selection
      DisableItem (EditMenu, ClearItem);

      EnableItem  (EditMenu, PasteItem);            // Enable Paste command

      ScrapDirty = true;                            // Mark scrap as dirty
      WindowDirty (true);                           // Mark window as dirty
```

```
   }  /* end DoCut */

//-----------------------------------------------------------------------------------------------------

void DoCopy (void)

   // Handle Copy command.

   {
      ScrollToSelection ();                        // Make sure selection is visible

      TECopy (TheText);                            // Copy the selection

      EnableItem (EditMenu, PasteItem);            // Enable Paste command

      ScrapDirty = true;                           // Mark scrap as dirty

   }  /* end DoCopy */

//-----------------------------------------------------------------------------------------------------

void DoPaste (void)

   // Handle Paste command.

   {
      ScrollToSelection ();                        // Make sure selection is visible

      TEPaste (TheText);                           // Paste the scrap

      AdjustScrollBar ();                          // Adjust scroll bar to length of text
      AdjustText ();                               // Adjust text to match scroll bar
      ScrollToSelection ();                        // Keep selection visible

      DisableItem (EditMenu, CutItem);             // Disable menu items that operate
      DisableItem (EditMenu, CopyItem);            //    on a nonempty selection
      DisableItem (EditMenu, ClearItem);

      WindowDirty (true);                          // Mark window as dirty

   }  /* end DoPaste */

//-----------------------------------------------------------------------------------------------------

void DoClear (void)

   // Handle Clear command.

   {
      ScrollToSelection ();                        // Make sure selection is visible

      TEDelete (TheText);                          // Delete the selection

      AdjustScrollBar ();                          // Adjust scroll bar to length of text
```

```
      AdjustText ();                                  // Adjust text to match scroll bar
      ScrollToSelection ();                           // Keep insertion point visible

      DisableItem (EditMenu, CutItem);                // Disable menu items that operate
      DisableItem (EditMenu, CopyItem);               //    on a nonempty selection
      DisableItem (EditMenu, ClearItem);

      WindowDirty (true);                             // Mark window as dirty

   }  /* end DoClear */

//-----------------------------------------------------------------------------------------------------------

void DoSysClick (WindowPtr whichWindow)

   // Handle mouse-down event in system window.

   {
      SystemClick (&TheEvent, whichWindow);           // Pass event to Toolbox for handling}

   }  /* end DoSysClick */

//-----------------------------------------------------------------------------------------------------------

void DoContent (WindowPtr whichWindow)

   // Handle mouse-down event in content region of active window.

   {
      Point          thePoint;                        // Location of mouse click in window coordinates
      ControlHandle  theControl;                      // Handle to control
      short          thePart;                         // Part of control where mouse was pressed
      Rect           textRect;                        // Window's text rectangle


      if ( whichWindow != FrontWindow() )                     // Is it an inactive window?
         SelectWindow (whichWindow);                          // If so, just activate it

      else
         {
            thePoint = TheEvent.where;                        // Get point in screen coordinates
            GlobalToLocal (&thePoint);                        // Convert to window coordinates

            thePart = FindControl (thePoint, whichWindow, &theControl); // Was mouse pressed in a control?

            if ( theControl == TheScrollBar )                 // Was it in the scroll bar?
               DoScroll (thePart, thePoint);                  // Go scroll the window

            else
               {
                  textRect = (**TheText).viewRect;            // Get text rectangle
                  if ( (theControl == nil)                    // Not in a control?
                        && ( PtInRect(thePoint, &textRect) ) ) // In text rectangle?
                     DoSelect (thePoint);                     // Go handle text selection
```

```
            }  /* end else */

         }  /* end else */

   }  /* end DoContent */

//--------------------------------------------------------------------------------------------------------

void DoScroll (short thePart, Point thePoint)

   // Handle mouse-down event in scroll bar.

   {
      if ( thePart == inThumb )                          // Dragging the indicator?

         {
            TrackControl (TheScrollBar, thePoint, nil);     // Track mouse with no action procedure
            AdjustText ();                                  // Adjust text to new setting

         }  /* end if ( thePart == inThumb ) */

      else

         TrackControl (TheScrollBar, thePoint, ProcPtr(ScrollText)); // Track mouse with continuous scroll

   }  /* end DoScroll */

//--------------------------------------------------------------------------------------------------------

pascal void ScrollText (ControlHandle theControl, short thePart)

   // Scroll text within window.

   {
      short  windowHeight;                          // Height of text rectangle in pixels
      short  windowLines;                           // Height of text rectangle in lines
      short  delta;                                 // Amount to scroll by, in lines
      short  oldValue;                              // Previous setting of scroll bar


      windowHeight = (**TheText).viewRect.bottom - (**TheText).viewRect.top;  // Find height of text rectangle
      windowLines  = windowHeight / (**TheText).lineHeight;                   // Convert to lines

      switch ( thePart )
         {
            case inUpButton:
               delta = -1;                          // Scroll up one line at a time
               break;

            case inDownButton:
               delta = +1;                          // Scroll down one line at a time
               break;
```

```
            case inPageUp:
               delta = 1 - windowLines;              // Scroll up by height of text rectangle
               break;

            case inPageDown:
               delta = windowLines - 1;              // Scroll down by height of text rectangle
               break;

            default:
               break;

         }  /* end switch ( thePart ) */

      if ( thePart != 0 )                            // Is mouse still in the original part?
         {
            oldValue = GetCtlValue (theControl);     // Get old setting
            SetCtlValue (theControl, oldValue + delta);  // Adjust by scroll amount

            AdjustText ();                           // Scroll text to match new setting

         }  /* end if (thePart != 0) */

   }  /* end ScrollText */

//-------------------------------------------------------------------------------------------------

void AdjustText (void)

   // Adjust text within window to match scroll bar setting.

   {
      short  viewTop;                                // Top of view (clipping) rectangle
      short  destTop;                                // Top of destination (wrapping) rectangle
      short  topLine;                                // First line currently visible in window
      short  heightPerLine;                          // Height of each text line in pixels
      short  oldScroll;                              // Old text offset in pixels
      short  newScroll;                              // New text offset in pixels


      viewTop       = (**TheText).viewRect.top;      // Get top of clip
      destTop       = (**TheText).destRect.top;      // Get top of wrap
      topLine       = GetCtlValue(TheScrollBar);     // Get current top line
      heightPerLine = (**TheText).lineHeight;        // Get height per line

      oldScroll = viewTop - destTop;                 // Find current offset
      newScroll = topLine * heightPerLine;           // Scroll bar gives new offset

      if ( oldScroll != newScroll )                  // Any difference?
         TEScroll (0, (oldScroll - newScroll), TheText); // Scroll by difference

   }  /* end AdjustText */

//-------------------------------------------------------------------------------------------------
```

```
pascal Boolean AutoScroll (void)

   // Handle automatic scrolling during text selection.

   {
      Point      mousePoint;                          // Mouse location in local (window) coordinates
      Rect       textRect;                            // Active window's text rectangle
      RgnHandle  saveClip;                            // Original clipping region on entry


      saveClip = NewRgn ();                           // Create temporary region
      GetClip  (saveClip);                            // Set it to existing clipping region
      ClipRect ( &(TheWindow->portRect) );            // Clip to entire port rectangle

         GetMouse (&mousePoint);                      // Find mouse location
         textRect = (**TheText).viewRect;             // Get text rectangle

         if ( mousePoint.v < textRect.top )           // Above top of rectangle?
            ScrollText (TheScrollBar, inUpButton);    // Scroll up one line

         else if ( mousePoint.v > textRect.bottom )   // Below bottom of rectangle?
            ScrollText (TheScrollBar, inDownButton);  // Scroll down one line

         /* else do nothing */

      SetClip    (saveClip);                          // Restore original clipping region
      DisposeRgn (saveClip);                          // Dispose of temporary region

      return true;                                    // Continue tracking mouse

   }  /* end AutoScroll */

//-------------------------------------------------------------------------------------------------------

void DoSelect (Point thePoint)

   // Handle mouse-down event in text rectangle.

   {
      Boolean  extend;                                   // Extend existing selection (Shift-click)?


      extend = ( (TheEvent.modifiers & shiftKey) != 0);  // Shift key down?

      TEClick (thePoint, extend, TheText);               // Do text selection

      FixEditMenu ();                                    // Enable/disable menu items

   }  /* end DoSelect */

//-------------------------------------------------------------------------------------------------------

void FixEditMenu (void)
```

```
   // Enable/disable editing commands.

   {
      short  selFirst;                              // Character position at start of selection
      short  selLast;                               // Character position at end of selection


      DisableItem (EditMenu, UndoItem);             // Disable Undo command


      selFirst = (**TheText).selStart;              // Get start of selection
      selLast  = (**TheText).selEnd;                // Get end of selection

      if ( selFirst == selLast )                    // Is selection empty?
         {
            DisableItem (EditMenu, CutItem);        // Disable menu items that operate
            DisableItem (EditMenu, CopyItem);       //    on a nonempty selection
            DisableItem (EditMenu, ClearItem);

         }  /* end if ( selFirst == selLast ) */

      else
         {
            EnableItem (EditMenu, CutItem);         // Enable menu items that operate
            EnableItem (EditMenu, CopyItem);        //    on a nonempty selection
            EnableItem (EditMenu, ClearItem);

         }  /* end else */


      if ( TEGetScrapLen() == 0 )                   // Is scrap empty?
         DisableItem (EditMenu, PasteItem);         // Disable Paste command
      else
         EnableItem  (EditMenu, PasteItem);         // Enable Paste command

   }  /* end FixEditMenu */

//-----------------------------------------------------------------------------------------------------

void DoDrag (WindowPtr whichWindow)

   // Handle mouse-down event in drag region.

   {
      Rect  limitRect;                          // Limit rectangle for dragging


      SetRect    (&limitRect, 0, MenuBarHeight, ScreenWidth, ScreenHeight);   // Set limit rectangle
      InsetRect  (&limitRect, ScreenMargin, ScreenMargin);                    // Inset by screen margin

      DragWindow (whichWindow, TheEvent.where, &limitRect);                   // Let user drag the window

   }  /* end DoDrag */
```

```
//--------------------------------------------------------------------------------------------------

void DoGrow (WindowPtr whichWindow)

   // Handle mouse-down event in size region.

   {
      Rect   sizeRect;                              // Minimum and maximum dimensions of window
      long   newSize;                               // Coded representation of new dimensions
      short  newWidth;                              // New width of window
      short  newHeight;                             // New height of window


      SetRect (&sizeRect,                           // Set size rectangle
                  MinWidth,
                  MinHeight,
                  ScreenWidth,                      // Maximum width is full screen
                  (ScreenHeight - MenuBarHeight) );   // Maximum height is full screen minus menu bar

      newSize = GrowWindow (whichWindow, TheEvent.where, &sizeRect); // Let user drag size region

      if ( newSize != 0 )                           // Was size changed?
         {
            EraseRect ( &(whichWindow->portRect) );   // Clear window to white

            newWidth  = LoWord(newSize);            // Extract width from low word
            newHeight = HiWord(newSize);            // Extract height from high word
            SizeWindow (whichWindow, newWidth, newHeight, true);  // Adjust size of window

            InvalRect ( &(whichWindow->portRect) );   // Force update of window's contents

            FixScrollBar ();                        // Resize scroll bar
            FixText ();                             // Resize text rectangle

         }  /* end if ( newSize != 0 ) */

   }  /* end DoGrow */

//--------------------------------------------------------------------------------------------------

void FixScrollBar (void)

   // Resize window's scroll bar.

   {
      HideControl (TheScrollBar);                   // Hide scroll bar

      MoveControl (TheScrollBar,                    // Move top-left corner
                   TheWindow->portRect.right
                        - (SBarWidth - 1),          // Allow for 1-pixel overlap at right
                   -1);                             // Overlap window top by 1 pixel

      SizeControl (TheScrollBar,                    // Adjust bottom-right corner
                   SBarWidth,
```

```
                (TheWindow->portRect.bottom + 1)
                     - (TheWindow->portRect.top - 1)
                     - (SBarWidth - 1) );          // Allow room for size box

      ShowControl (TheScrollBar);                  // Redisplay scroll bar

      ValidRect ( &( (**TheScrollBar).contrlRect ) ); // Avoid updating again

   }  /* end FixScrollBar */

//--------------------------------------------------------------------------------------------------

void FixText (void)

   // Resize window's text rectangle.

   {
      short  topLine;                              // First line visible in window
      short  firstChar;                            // Character position of first character in window
      short  heightPerLine;                        // Height of each line in pixels
      Rect  *textRect;                             // Pointer to text rectangle


      SetCursor ( &(**Watch) );                    // Indicate delay

      topLine       = GetCtlValue (TheScrollBar);     // Get previous first line
      firstChar     = (**TheText).lineStarts[topLine];   // Find first character previously visible
      heightPerLine = (**TheText).lineHeight;         // Get line height

      textRect  = &( (**TheText).destRect );          // Point to wrapping rectangle
      *textRect = TheWindow->portRect;                // Wrap text to window's port rectangle
      (*textRect).right  -= (SBarWidth - 1);          // Exclude scroll bar, allowing 1-pixel overlap
      (*textRect).bottom -= (SBarWidth - 1);          // Leave space for scroll bar at bottom
      (*textRect).bottom /= heightPerLine;            // Truncate to a whole
      (*textRect).bottom *= heightPerLine;            //    number of lines

      (**TheText).viewRect = *textRect;               // Clip to same rectangle
      InsetRect (textRect, TextMargin, TextMargin);   // Inset wrap by text margin

      TECalText (TheText);                            // Recalibrate line starts
      AdjustScrollBar ();                             // Adjust scroll bar to new length

      ScrollCharacter (firstChar, false);             // Scroll same character to top of window

   }  /* end FixText */

//--------------------------------------------------------------------------------------------------

void DoGoAway (WindowPtr whichWindow)

   // Handle mouse-down event in close region.

   {
      if ( TrackGoAway(whichWindow, TheEvent.where) )    // Track mouse in close region and
```

```
         DoClose ();                                    //    close window if necessary

   }  /* end DoGoAway */

//--------------------------------------------------------------------------------------------------------------

void DoZoom (WindowPtr whichWindow, short inOrOut)

   // Handle mouse-down event in zoom region.

   {
      if ( TrackBox(whichWindow, TheEvent.where, inOrOut) )    // Track mouse in zoom region
         {
            EraseRect  ( &(whichWindow->portRect) );           // Clear window to white

            ZoomWindow (whichWindow, inOrOut, false);          // Zoom the window

            InvalRect  ( &(whichWindow->portRect) );           // Force update of window's contents

            FixScrollBar ();                                   // Resize scroll bar
            FixText ();                                        // Resize text rectangle

         } /* end if ( TrackBox(whichWindow, TheEvent.where, inOrOut) ) */

   }  /* end DoZoom */

//--------------------------------------------------------------------------------------------------------------

void DoKeystroke (void)

   // Handle keystroke.

   {
      unsigned char  ch;                              // Character that was typed
      long           menuChoice;                      // Menu ID and item number for Command combination


      ch = TheEvent.message & charCodeMask;           // Extract character code

      if ( (TheEvent.modifiers & cmdKey) != 0 )       // Command key down?
         {
            if ( TheEvent.what != autoKey )           // Ignore repeats
               {
                  menuChoice = MenuKey(ch);           // Get menu equivalent
                  DoMenuChoice (menuChoice);          // Handle as menu choice

               }  /* end if ( TheEvent.what != autoKey ) */

         }  /* end if ( (TheEvent.modifiers & cmdKey) != 0 ) */

      else
         DoTyping (ch);                               // Handle as normal character

   }  /* end DoKeystroke */
```

```
//--------------------------------------------------------------------------------------------------------

void DoTyping (unsigned char ch)

   // Handle character typed from keyboard.

   {
      if ( TheText == nil )                        // Is screen empty?
         {
            SysBeep(1);                            // Just beep
            return;                                //    and exit
         }  /* end if ( TheText == nil ) */

      ScrollToSelection ();                        // Make sure insertion point is visible

      TEKey (ch, TheText);                         // Process character

      AdjustScrollBar ();                          // Adjust scroll bar to length of text
      AdjustText ();                               // Adjust text to match scroll bar
      ScrollToSelection ();                        // Keep insertion point visible

      DisableItem (EditMenu, CutItem);             // Disable menu items that operate
      DisableItem (EditMenu, CopyItem);            //    on a nonempty selection
      DisableItem (EditMenu, ClearItem);

      WindowDirty (true);                          // Mark window as dirty

   }  /* end DoTyping */

//--------------------------------------------------------------------------------------------------------

void DoUpdate (void)

   // Handle update event.

   {
      GrafPtr    savePort;                         // Pointer to previous current port
      WindowPtr  whichWindow;                      // Pointer to window to be updated
      long       theRefCon;                        // Window's reference constant
      WDHandle   theData;                          // Handle to window's data record
      TEHandle   theEditRec;                       // Handle to window's edit record
      Rect       theViewRect;                      // Edit record's view rectangle


      GetPort (&savePort);                         // Save previous port

         whichWindow = WindowPtr(TheEvent.message);// Convert long integer to pointer
         SetPort (whichWindow);                    // Make window the current port

         BeginUpdate (whichWindow);                // Restrict visible region to update region

            EraseRect ( &(whichWindow->portRect) );// Clear update region
```

```
            DrawGrowIcon (whichWindow);              // Redraw size box
            DrawControls (whichWindow);              // Redraw scroll bar

            theRefCon   = GetWRefCon(whichWindow);   // Get reference constant
            theData     = WDHandle(theRefCon);       // Convert to data handle
            theEditRec  = (**theData).editRec;       // Get edit record
            theViewRect = (**theEditRec).viewRect;   // Get view rectangle
            TEUpdate (&theViewRect, theEditRec);     // Redraw the text

         EndUpdate (whichWindow);                    // Restore original visible region

      SetPort (savePort);                            // Restore original port

   }  /* end DoUpdate */

//--------------------------------------------------------------------------------------------------------

void DoActivate (void)

   // Handle activate (or deactivate) event.

   {
      WindowPtr  whichWindow;                        // Pointer to the window


      whichWindow = WindowPtr( TheEvent.message );   // Convert long integer to pointer

      if ( (TheEvent.modifiers & activeFlag) != 0 )  // Test activate/deactivate bit
         ActWindow   (whichWindow);                  // Activate window
      else
         DeactWindow (whichWindow);                  // Deactivate window

   }  /* end DoActivate */

//--------------------------------------------------------------------------------------------------------

void ActWindow (WindowPtr whichWindow)

   // Activate window.

   {
      const short  active = 0;                       // Highlighting code for active scroll bar

      long      theRefCon;                           // Window's reference constant
      WDHandle  theData;                             // Handle to window data record
      short     theFile;                             // Reference number of window's file


      theRefCon = GetWRefCon(whichWindow);           // Get reference constant
      theData   = WDHandle(theRefCon);               // Convert to data handle

      TheWindow    = whichWindow;                    // Set global pointers/handles
      TheScrollBar = (**theData).scrollBar;
      TheText      = (**theData).editRec;
```

```
      SetPort (TheWindow);                          // Make window the current port

      DrawGrowIcon  (TheWindow);                    // Highlight size box
      HiliteControl (TheScrollBar, active);         // Activate scroll bar
      TEActivate    (TheText);                      // Highlight selection

      if ( (TheEvent.modifiers & ChangeFlag) != 0 )  // Coming from a system window?
         ReadDeskScrap ();                          // Copy desk scrap to Toolbox scrap

      FixEditMenu ();                               // Enable/disable editing commands

      EnableItem (FileMenu, SaveAsItem);            // Enable Save As... command
      if ( (**theData).dirty )                      // Is document dirty?
         {
            EnableItem (FileMenu, SaveItem);        // Enable Save command

            theFile = (**theData).fileNumber;       // Get file reference number
            if ( theFile != 0 )                     // Is there a file to revert to?
               EnableItem (FileMenu, RevertItem);   // Enable Revert command

         }  /* end if ( (**theData).dirty ) */

   }  /* end ActWindow */

//--------------------------------------------------------------------------------------------------------

void DeactWindow (WindowPtr whichWindow)

   // Deactivate window.

   {
      const short  inactive = 255;                  // Highlighting code for inactive scroll bar

      long      theRefCon;                          // Window's reference constant
      WDHandle  theData;                            // Handle to window data record


      theRefCon = GetWRefCon(whichWindow);          // Get reference constant
      theData   = WDHandle(theRefCon);              // Convert to data handle

      TheWindow    = nil;                           // Clear global pointers/handles
      TheScrollBar = nil;
      TheText      = nil;

      SetPort (whichWindow);                        // Make window the current port

      DrawGrowIcon  (whichWindow);                  // Unhighlight size box
      HiliteControl ((**theData).scrollBar, inactive);   // Deactivate scroll bar
      TEDeactivate  ((**theData).editRec);          // Unhighlight selection

      if ( (TheEvent.modifiers & ChangeFlag) != 0 )  // Exiting to a system window?
         {
            WriteDeskScrap ();                      // Copy Toolbox scrap to desk scrap
```

```
            EnableItem (EditMenu, UndoItem);          // Enable standard editing commands
            EnableItem (EditMenu, CutItem);           //    for desk accessory
            EnableItem (EditMenu, CopyItem);
            EnableItem (EditMenu, PasteItem);
            EnableItem (EditMenu, ClearItem);

         }  /* end if ( (TheEvent.modifiers & ChangeFlag) != 0 ) */

      DisableItem (FileMenu, SaveItem);               // Disable filing commands for desk
      DisableItem (FileMenu, SaveAsItem);             //    accessory or empty desk
      DisableItem (FileMenu, RevertItem);

   }  /* end DeactWindow */

//--------------------------------------------------------------------------------------------------------

void Finalize (void)

   // Do one-time-only finalization.

   {
      WriteDeskScrap ();                              // Copy Toolbox scrap to desk scrap

      SetEventMask (OldMask);                         // Restore original value of system event mask

   }  /* end Finalize */

//--------------------------------------------------------------------------------------------------------

void WindowDirty (Boolean isDirty)

   // Mark window dirty or clean.

   {
      long      theRefCon;                            // Window's reference constant
      WDHandle  theData;                              // Handle to window data record
      short     theFile;                              // Reference number of window's file


      theRefCon = GetWRefCon(TheWindow);              // Get reference constant
      theData   = WDHandle(theRefCon);                // Convert to data handle

      (**theData).dirty = isDirty;                    // Set flag in data record

      if ( isDirty )                                  // Is window becoming dirty or clean?
         {
            EnableItem (FileMenu, SaveItem);          // Enable Save command

            theFile = (**theData).fileNumber;         // Get file reference number
            if ( theFile != 0 )                       // Is window associated with a file?
               EnableItem (FileMenu, RevertItem);     // Enable Revert command

         }  /* end if ( isDirty ) */
```

```
      else
         {
            DisableItem (FileMenu, SaveItem);         // Disable menu items
            DisableItem (FileMenu, RevertItem);

         }  /* end else */

   }  /* end WindowDirty */

//--------------------------------------------------------------------------------------------------------

void AdjustScrollBar (void)

   // Adjust scroll bar to length of document.

   {
      const short  active   =   0;                   // Highlighting code for active scroll bar
      const short  inactive = 255;                   // Highlighting code for inactive scroll bar

      short  windowTop;                              // Top of window's text rectangle
      short  windowBottom;                           // Bottom of window's text rectangle
      short  heightPerLine;                          // Height of each text line in pixels
      short  windowHeight;                           // Height of text rectangle in pixels
      short  windowLines;                            // Height of text rectangle in lines
      short  textLines;                              // Total lines of text
      short  maxTop;                                 // Maximum value for top line in window


      windowTop     = (**TheText).viewRect.top;      // Get top of rectangle
      windowBottom  = (**TheText).viewRect.bottom;   // Get bottom of rectangle
      textLines     = (**TheText).nLines;            // Get total lines
      heightPerLine = (**TheText).lineHeight;        // Get height per line

      windowHeight = windowBottom - windowTop;       // Find height of text rectangle
      windowLines  = windowHeight / heightPerLine;   // Convert to lines
      maxTop       = textLines - windowLines;        // Avoid white space at bottom

      if ( maxTop <= 0 )                             // Is text smaller than window?
         {
            maxTop = 0;                              // Show all of text
            HiliteControl (TheScrollBar, inactive);  // Disable scroll bar

         }  /* end if ( maxTop <= 0 ) */
      else
         HiliteControl (TheScrollBar, active);       // Enable scroll bar

      SetCtlMax (TheScrollBar, maxTop);              // Adjust range of scroll bar

   }  /* end AdjustScrollBar */

//--------------------------------------------------------------------------------------------------------

void ScrollToSelection (void)
```

```
   // Scroll current selection into view.

   {
      short  windowTop;                          // Top of window's text rectangle
      short  windowBottom;                       // Bottom of window's text rectangle
      short  heightPerLine;                      // Height of each text line in pixels
      short  windowHeight;                       // Height of text rectangle in pixels
      short  windowLines;                        // Height of text rectangle in lines
      short  topLine;                            // First line visible in window
      short  bottomLine;                         // First line beyond bottom of window
      short  topChar;                            // First character position visible in window
      short  bottomChar;                         // First character position beyond bottom of window
      short *lineBreaks;                            // Pointer to array of line starts
      short  selFirst;                           // Character position at start of selection
      short  selLast;                            // Character position at end of selection


      windowTop     = (**TheText).viewRect.top;       // Get top of rectangle
      windowBottom  = (**TheText).viewRect.bottom;    // Get bottom of rectangle
      heightPerLine = (**TheText).lineHeight;         // Get height per line

      windowHeight = windowBottom - windowTop;        // Find height of text rectangle in pixels
      windowLines  = windowHeight / heightPerLine;    // Convert to lines

      topLine    = GetCtlValue(TheScrollBar);         // Get current top line
      bottomLine = topLine + windowLines;             // Find line beyond bottom

      lineBreaks = (**TheText).lineStarts;            // Point to start of line array
      topChar    = lineBreaks[topLine];               // Find first visible character
      bottomChar = lineBreaks[bottomLine];            // Find first character beyond bottom

      selFirst = (**TheText).selStart;                // Get start of selection
      selLast  = (**TheText).selEnd;                  // Get end of selection

      if ( GetCtlMax(TheScrollBar) == 0 )             // Not enough text to fill the window?
         AdjustText ();                               // Start of text to top of window
      else if ( selLast < topChar )                   // Whole selection above window top?
         ScrollCharacter (selFirst, false);           // Start of selection to top of window
      else if ( selFirst >= bottomChar )              // Whole selection below window bottom?
         ScrollCharacter (selLast, true);             // End of selection to bottom of window

   }  /* end ScrollToSelection */

//-----------------------------------------------------------------------------------------------------------

void ScrollCharacter (short theCharacter, Boolean toBottom)

   // Scroll character into view.

   {
      short  windowTop;                              // Top of window's text rectangle
      short  windowBottom;                           // Bottom of window's text rectangle
      short  heightPerLine;                          // Height of each text line in pixels
      short  windowHeight;                           // Height of text rectangle in pixels
```

```
      short  windowLines;                              // Height of text rectangle in lines
      short *lineBreaks;                               // Pointer to array of line starts
      short  thisLine;                                 // Number of line containing character


      lineBreaks = (**TheText).lineStarts;             // Point to start of line array
      for ( (thisLine = 0); (lineBreaks[thisLine + 1] <= theCharacter); (thisLine++) )
         {  /* Search for line containing character */ }

      if ( toBottom )                                  // Scrolling to bottom of window?
         {
            windowTop     = (**TheText).viewRect.top;    // Get top of rectangle
            windowBottom  = (**TheText).viewRect.bottom; // Get bottom of rectangle
            heightPerLine = (**TheText).lineHeight;      // Get height per line

            windowHeight = windowBottom - windowTop;     // Find height of text rectangle in pixels
            windowLines  = windowHeight / heightPerLine; // Convert to lines
            thisLine -= (windowLines - 1);               // Offset for window height

         }  /* end if ( toBottom ) */

      SetCtlValue (TheScrollBar, thisLine);            // Adjust setting of scroll bar
      AdjustText ();                                   // Scroll text to match new setting

   }  /* end ScrollCharacter */

//-------------------------------------------------------------------------------------------------------------

void ReadDeskScrap (void)

   // Read desk scrap into Toolbox scrap.

   {
      PScrapStuff  scrapInfo;                       // Pointer to scrap information record
      long         scrapLength;                     // Length of desk text scrap in bytes
      long         ignore;                          // Dummy variable for scrap offset
      OSErr        result;                          // Result code from scrap transfer


      scrapInfo = InfoScrap();                      // Get scrap info

      if ( ScrapCompare != scrapInfo->scrapCount )  // Has scrap count changed?
         {
            scrapLength = GetScrap(nil, 'TEXT', &ignore);   // Check desk scrap for a text item

            if ( scrapLength >= 0 )                 // Is there a text item?
               {
                  result = TEFromScrap();           // Transfer desk scrap to Toolbox scrap
                  if ( result != noErr )            // Was there an error?
                     scrapLength = result;          // Make sure scrap length is negative

               }  /* end if ( scrapLength >= 0 ) */

            if ( scrapLength > 0 )                  // Was scrap nonempty?
```

```
            EnableItem (EditMenu, PasteItem);      // Enable Paste command
         else
            {
               TESetScrapLen (0);                  // Mark Toolbox scrap as empty
               DisableItem (EditMenu, PasteItem);  // Disable Paste command

            }  /* end else */

         scrapInfo    = InfoScrap();               // Get scrap info
         ScrapCompare = scrapInfo->scrapCount;     // Save scrap count for later comparison

      }  /* end if ( ScrapCompare() <> scrapInfo->scrapCount ) */

   }  /* end ReadDeskScrap */

//-----------------------------------------------------------------------------------------------------

void WriteDeskScrap (void)

   // Write Toolbox scrap to desk scrap.

   {
      PScrapStuff  scrapInfo;                     // Pointer to scrap information record
      long         scrapResult;                   // Result code from ZeroScrap
      OSErr        teResult;                      // Result code from scrap transfer


      if ( ScrapDirty )                           // Has scrap changed since last read?
         {
            scrapResult = ZeroScrap();            // Change scrap count
            if ( scrapResult == noErr )           // Was there an error?
               {
                  scrapInfo    = InfoScrap();         // Get scrap info
                  ScrapCompare = scrapInfo->scrapCount;  // Save new scrap count for comparison

               }  /* end if ( scrapResult == noErr ) */

            teResult = TEToScrap ();              // Transfer Toolbox scrap to desk scrap

            ScrapDirty = false;                   // Toolbox and desk scraps now agree

         }  /* end if ( ScrapDirty ) */

   }  /* end WriteDeskScrap */

//-----------------------------------------------------------------------------------------------------

void IOCheck (OSErr resultCode)

   // Check for I/O error.

   {
      short   alertID;                            // Resource ID of alert
      Str255  errorString;                        // Error code in string form
```

```
   if ( resultCode == noErr )                    // Just return if no error
      return;

   switch ( resultCode )
      {
         case opWrErr:                           // File already open?
            alertID = OpWrID;                    // Use Already Open alert
            break;

         /* Insert code here to handle any other specific errors */

         default:
            alertID = IOErrID;                          // Use general I/O Error alert
            NumToString (resultCode, errorString);      // Convert error code to a string
            ParamText (errorString, "\p", "\p", "\p");  // Substitute into text of alert
            break;

      }  /* end switch ( resultCode ) */

   InitCursor ();                                // Restore normal cursor
   StopAlert (alertID, nil);                     // Post alert

   Quitting  = false;                            // Cancel Quit command, if any
   ErrorFlag = true;                             // Force exit to main event loop

}  /* end IOCheck */
```

Resource Description File

The resource description file, **`MiniEdit.r`**, was generated by the DeRez resource decompiler from the resources I built onscreen with the interactive resource editor ResEdit.

```
//
//                                  MiniEdit
//                     Example Macintosh application program
//                        S. Chernicoff    24 October 1994
//
//
//    Multiple-window text editor to illustrate event-driven program structure
//
//
//    MiniEdit.r - Rez source file


#include "systypes.r"
#include "types.r"

data 'MENU' (1, "Apple") {
   $"0001 0000 0000 0000 0000 FFFF FFFB 0114"           /* ..........˜˜˜˚.. */
```

```
   $"0F41 626F 7574 204D 696E 6945 6469 74C9"          /* .About MiniEdit… */
   $"0000 0000 012D 0000 0000 00"                      /* .....-..... */
};

data 'MENU' (2, "File") {
   $"0002 0000 0000 0000 0000 FFFF F007 0446"          /* ..........ˇˇ..F */
   $"696C 6503 4E65 7700 4E00 0005 4F70 656E"          /* ile.New.N...Open */
   $"C900 4F00 0005 436C 6F73 6500 5700 0001"          /* ….O...Close.W... */
   $"2D00 0000 0004 5361 7665 0053 0000 0853"          /* -.....Save.S...S */
   $"6176 6520 4173 C900 4100 000F 5265 7665"          /* ave As….A...Reve */
   $"7274 2074 6F20 5361 7665 6400 5200 0001"          /* rt to Saved.R... */
   $"2D00 0000 000B 5061 6765 2053 6574 7570"          /* -.....Page Setup */
   $"C900 5500 0006 5072 696E 74C9 0050 0000"          /* ….U...Print….P.. */
   $"012D 0000 0000 0451 7569 7400 5100 0000"          /* .-.....Quit.Q... */
};

data 'MENU' (3, "Edit") {
   $"0003 0000 0000 0000 0000 FFFF FF01 0445"          /* ..........ˇˇˇ..E */
   $"6469 7404 556E 646F 005A 0000 012D 0000"          /* dit.Undo.Z...-.. */
   $"0000 0343 7574 0058 0000 0443 6F70 7900"          /* ...Cut.X...Copy. */
   $"4300 0005 5061 7374 6500 5600 0001 2D00"          /* C...Paste.V...-. */
   $"0000 0005 436C 6561 7200 4200 0000"               /* ....Clear.B... */
};

data 'ALRT' (1000, "About MiniEdit") {
   $"003C 005A 00DC 01A4 03E8 4444"                    /* .<.Z.‹.§.ËDD */
};

data 'ALRT' (1001, "Save") {
   $"003C 0078 00B8 0166 03E9 5555"                    /* .<.x.∏.f.ÈUU */
};

data 'ALRT' (1002, "Revert") {
   $"003C 0057 009A 01A9 03EA 5555"                    /* .<.W.ö.©.ÍUU */
};

data 'ALRT' (1003, "Can’t Print") {
   $"003C 0064 00A4 019A 03EB 4444"                    /* .<.d.§.ö.ÎDD */
};

data 'ALRT' (1004, "Wrong Type") {
   $"003C 0064 00AC 019C 03EC 5555"                    /* .<.d.¨.ú.ÌUU */
};

data 'ALRT' (1005, "File Too Long") {
   $"003C 0070 00AA 0190 03ED 5555"                    /* .<.p.™.ê.ÌUU */
};

data 'ALRT' (1006, "Already Open") {
   $"003C 005C 008C 01A4 03EE 5555"                    /* .<.\.å.§.ÓUU */
};

data 'ALRT' (1007, "I/O Error") {
   $"003C 00A5 008C 015B 03EF 5555"                    /* .<.•.å.[.ÔUU */
```

```
};

data 'DITL' (1000, "About MiniEdit") {
   $"0001 0000 0000 0082 0082 0096 00C8 0408"          /* .......Ç.Ç.ñ.».. */
   $"436F 6E74 696E 7565 0000 0000 000A 0014"          /* Continue........ */
   $"007E 0136 88AC 2020 2020 2020 2020 2020"          /* .~.6à¨          */
   $"2020 2020 2020 2020 2020 2020 2020 2020"          /*                 */
   $"4D69 6E69 4564 6974 2043 2B2B 0D0D D24D"          /* MiniEdit C++¬¬“M */
   $"6163 696E 746F 7368 2052 6576 6561 6C65"          /* acintosh Reveale */
   $"64D3 2065 7861 6D70 6C65 2061 7070 6C69"          /* d” example appli */
   $"6361 7469 6F6E 0D0D 2020 2020 2020 2020"          /* cation¬¬        */
   $"2020 2020 2020 2020 2020 2020 2020 2020"          /*                 */
   $"2020 432B 2B20 7665 7273 696F 6E0D 0D20"          /*   C++ version¬¬  */
   $"532E 2043 6865 726E 6963 6F66 6620 2020"          /* S. Chernicoff    */
   $"2020 2020 2020 2020 2020 2020 2020 2020"          /*                 */
   $"2020 2032 3420 4F63 746F 6265 7220 3139"          /*    24 October 19 */
   $"3934"                                             /* 94 */
};

data 'DITL' (1001, "Save") {
   $"0003 0000 0000 0040 0014 0054 005A 0404"          /* .......@...T.Z.. */
   $"5361 7665 0000 0000 005E 0014 0072 005A"          /* Save.....^...r.Z */
   $"0407 4469 7363 6172 6400 0000 0000 005E"          /* ..Discard......^ */
   $"009B 0072 00E1 0406 4361 6E63 656C 0000"          /* .õ.r.·..Cancel.. */
   $"0000 000A 003E 003C 00E1 881E 5361 7665"          /* .....>.<.·à.Save */
   $"2066 696C 6520 D25E 30D3 0D62 6566 6F72"          /*  file “^0”¬befor */
   $"6520 636C 6F73 696E 673F"                         /* e closing? */
};

data 'DITL' (1002, "Revert") {
   $"0002 0000 0000 0040 0014 0054 005A 0406"          /* .......@...T.Z.. */
   $"5265 7665 7274 0000 0000 0040 00F8 0054"          /* Revert.....@.¯.T */
   $"013E 0406 4361 6E63 656C 0000 0000 000A"          /* .>..Cancel...... */
   $"003C 003C 013E 8833 5265 7665 7274 2074"          /* .<.<.>à3Revert t */
   $"6F20 6D6F 7374 2072 6563 656E 746C 7920"          /* o most recently  */
   $"7361 7665 6420 7665 7273 696F 6E20 6F66"          /* saved version of */
   $"2066 696C 6520 D25E 30D3 3F00"                    /*  file “^0”?. */
};

data 'DITL' (1003, "Can’t Print") {
   $"0001 0000 0000 0046 00D4 0058 011A 0404"          /* .......F.‘.X.... */
   $"5175 6974 0000 0000 000A 003C 002A 0121"          /* Quit.......<.*.! */
   $"8834 536F 7272 792C 2074 6869 7320 7665"          /* à4Sorry, this ve */
   $"7273 696F 6E20 6F66 204D 696E 6945 6469"          /* rsion of MiniEdi */
   $"7420 0D63 616E D574 2070 7269 6E74 2061"          /* t ¬can’t print a */
   $"2066 696C 652E"                                   /*  file. */
};

data 'DITL' (1004, "Wrong Type") {
   $"0001 0000 0000 0052 00DE 0066 0124 0402"          /* .......R.fi.f.$.. */
   $"4F4B 0000 0000 000A 003E 004E 0124 884B"          /* OK.......>.N.$àK */
   $"536F 7272 792C 204D 696E 6945 6469 7420"          /* Sorry, MiniEdit  */
   $"776F 726B 7320 7769 7468 2074 6578 7420"          /* works with text  */
   $"6669 6C65 7320 6F6E 6C79 2E20 2043 616E"          /* files only.  Can */
```

```
   $"2774 2072 6561 6420 6F72 2077 7269 7465"          /* 't read or write */
   $"2066 696C 6520 D25E 30D3 2E00"                    /*  file “^0”.. */
};

data 'DITL' (1005, "File Too Long") {
   $"0001 0000 0000 0050 00CE 0064 0114 0406"          /* .......P.Œ.d.... */
   $"4361 6E63 656C 0000 0000 000A 003E 004C"          /* Cancel.......>.L */
   $"010C 8831 536F 7272 792C 2066 696C 6520"          /* ..à1Sorry, file  */
   $"D25E 30D3 0D69 7320 746F 6F20 6C6F 6E67"          /* “^0”¬is too long */
   $"2066 6F72 204D 696E 6945 6469 7420 746F"          /*  for MiniEdit to */
   $"2072 6561 642E"                                   /*  read. */
};

data 'DITL' (1006, "Already Open") {
   $"0001 0000 0000 0032 00EE 0046 0134 0406"          /* .......2.Ó.F.4.. */
   $"4361 6E63 656C 0000 0000 000A 003E 001C"          /* Cancel.......>.. */
   $"0134 8825 536F 7272 792C 2063 616E 2774"          /* .4à%Sorry, can't */
   $"206F 7065 6E20 7468 6520 7361 6D65 2066"          /*  open the same f */
   $"696C 6520 7477 6963 652E"                         /* ile twice. */
};

data 'DITL' (1007, "I/O Error") {
   $"0001 0000 0000 0032 005C 0046 00A2 0406"          /* .......2.\.F.¢.. */
   $"4361 6E63 656C 0000 0000 000A 003E 001C"          /* Cancel.......>.. */
   $"00A2 880C 492F 4F20 4572 726F 7220 5E30"          /* .¢à.I/O Error ^0 */
};

data 'WIND' (1000, "MiniEdit window template") {
   $"0046 0032 0116 015E 0008 0000 0100 0000"          /* .F.2...^........ */
   $"0000 0875 6E74 6974 6C65 64"                      /* ...untitled */
};

data 'STR ' (1000, "Default window title") {
   $"0875 6E74 6974 6C65 64"                           /* .untitled */
};

data 'CNTL' (1000, "Scroll bar template") {
   $"FFFF 011D 00C2 012D 0000 0100 0000 0000"          /* ˇˇ...¬.-........ */
   $"0010 0000 0000 012A"                              /* .......* */
};

data 'BNDL' (128, "MiniEdit bundle") {
   $"4D45 4454 0000 0001 4652 4546 0001 0000"          /* MEDT....FREF.... */
   $"0080 0001 0081 4943 4E23 0001 0000 0080"          /* .Ä...ÅICN#.....Ä */
   $"0001 0081"                                        /* ...Å */
};

data 'FREF' (128, "APPL") {
   $"4150 504C 0000 00"                                /* APPL... */
};

data 'FREF' (129, "TEXT") {
   $"5445 5854 0001 00"                                /* TEXT... */
};
```

```
data 'ICN#' (128, "MiniEdit APPL file icon") {
   $"FFFF FFFF 8000 0001 8000 0001 FFFF FFFF"          /* ˜˜˜˜Ä...Ä...˜˜˜˜ */
   $"D555 5555 AAAA AAAB FFFD 5555 A006 AAAB"           /* ’UUU™™™™´˜˜UU†.™´ */
   $"FFFD 7FFD A006 C00B E005 7FFD A006 C00B"          /* ˜˜.˝†.¿.‡..˝†.¿. */
   $"E7FF C00D A400 400B E7FF C00D A400 400B"          /* Á˜¿¬§.@.Á˜¿¬§.@. */
   $"E400 400D A400 400B FC00 400D AC00 7FFB"          /* ‰.@¬§.@.¸.@¬¨..° */
   $"D400 00D5 AC00 00AB D400 00D5 AC00 00AB"          /* ‘..’¨..´‘..’¨..´ */
   $"D400 00D5 AC00 00AB D400 00D5 AC00 00AB"          /* ‘..’¨..´‘..’¨..´ */
   $"D400 00D5 AFFF FFAB D555 5555 FFFF FFFF"          /* ‘..’Ø˜˜´’UUU˜˜˜˜ */
   $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"          /* ˜˜˜˜˜˜˜˜˜˜˜˜˜˜˜˜ */
   $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"          /* ˜˜˜˜˜˜˜˜˜˜˜˜˜˜˜˜ */
   $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"          /* ˜˜˜˜˜˜˜˜˜˜˜˜˜˜˜˜ */
   $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"          /* ˜˜˜˜˜˜˜˜˜˜˜˜˜˜˜˜ */
   $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"          /* ˜˜˜˜˜˜˜˜˜˜˜˜˜˜˜˜ */
   $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"          /* ˜˜˜˜˜˜˜˜˜˜˜˜˜˜˜˜ */
   $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"          /* ˜˜˜˜˜˜˜˜˜˜˜˜˜˜˜˜ */
   $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"          /* ˜˜˜˜˜˜˜˜˜˜˜˜˜˜˜˜ */
};

data 'ICN#' (129, "MiniEdit TEXT file icon", purgeable, preload) {
   $"0FFF FC00 0800 0600 0800 0500 0800 0480"          /* .˜¸.............Ä */
   $"08F8 0440 0800 0420 081F E7F0 0800 0010"          /* .¯.@... ..Á.... */
   $"081F E010 0800 0010 0803 FC10 0800 0010"          /* ..‡.......¸..... */
   $"081F 0010 0800 0010 081F FE10 0800 0010"          /* ..........˛..... */
   $"08FF E010 0800 0010 083C 0010 0800 0010"          /* .˜‡......<..... */
   $"080F C010 0800 0010 080F F810 0800 0010"          /* ..¿.......¯..... */
   $"08FC 0010 0800 0010 083F E010 0800 0010"          /* .¸.......?‡..... */
   $"0807 FE10 0800 0010 0800 0010 0FFF FFF0"          /* ..˛..........˜˜ */
   $"0FFF FC00 0FFF FE00 0FFF FF00 0FFF FF80"          /* .˜¸..˜˛..˜˜..˜˜Ä */
   $"0FFF FFC0 0FFF FFE0 0FFF FFF0 0FFF FFF0"          /* .˜˜¿.˜˜‡.˜˜.˜˜ */
   $"0FFF FFF0 0FFF FFF0 0FFF FFF0 0FFF FFF0"          /* .˜˜.˜˜.˜˜.˜˜ */
   $"0FFF FFF0 0FFF FFF0 0FFF FFF0 0FFF FFF0"          /* .˜˜.˜˜.˜˜.˜˜ */
   $"0FFF FFF0 0FFF FFF0 0FFF FFF0 0FFF FFF0"          /* .˜˜.˜˜.˜˜.˜˜ */
   $"0FFF FFF0 0FFF FFF0 0FFF FFF0 0FFF FFF0"          /* .˜˜.˜˜.˜˜.˜˜ */
   $"0FFF FFF0 0FFF FFF0 0FFF FFF0 0FFF FFF0"          /* .˜˜.˜˜.˜˜.˜˜ */
   $"0FFF FFF0 0FFF FFF0 0FFF FFF0 0FFF FFF0"          /* .˜˜.˜˜.˜˜.˜˜ */
   $"0FFF FFF0 0FFF FFF0 0FFF FFF0 0FFF FFF0"          /* .˜˜.˜˜.˜˜.˜˜ */
   $"0FFF FFF0 0FFF FFF0"                              /* .˜˜.˜˜ */
};

data 'MEDT' (0, "MiniEdit autograph") {
   $"514D 696E 6945 6469 740D 4578 616D 706C"          /* QMiniEdit¬Exampl */
   $"6520 4D61 6369 6E74 6F73 6820 6170 706C"          /* e Macintosh appl */
   $"6963 6174 696F 6E20 7072 6F67 7261 6D0D"          /* ication program¬ */
   $"532E 2043 6865 726E 6963 6F66 662C 2032"          /* S. Chernicoff, 2 */
   $"3420 4F63 746F 6265 7220 3139 3934"               /* 4 October 1994 */
};
```

951231-SC